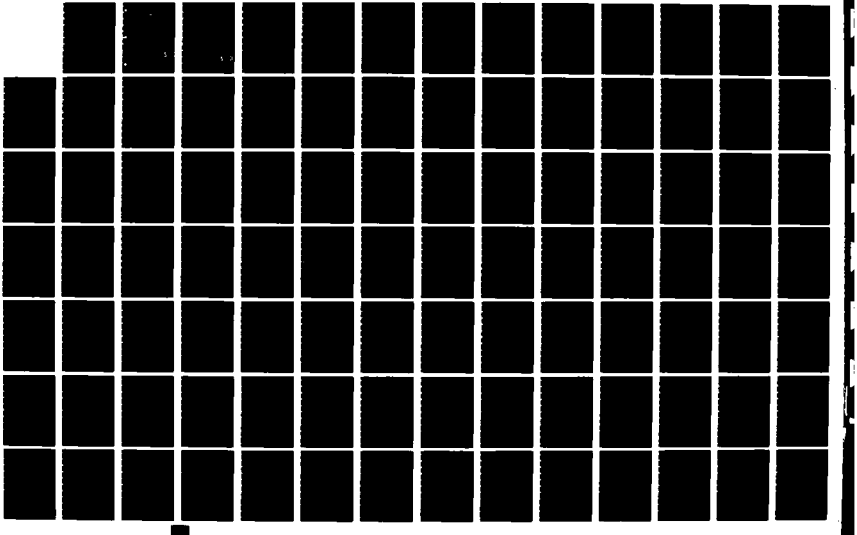
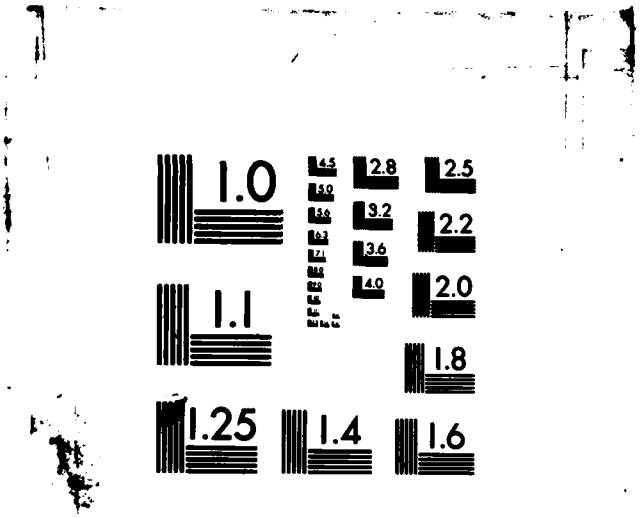
AD-A138 433 A CANDIDATE PROGRAMMING LANGUAGE(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING R JENNINGS DEC 83 AFIT/GA/EE/83D-1 1/2

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

A CANDIDATE PROGRAMMING LANGUAGE

THESIS

Richard Jennings
Capt USAF

AFIT/GA/EE/83D-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GA/EE/83D-1

A CANDIDATE PROGRAMMING LANGUAGE

THESIS

Richard Jennings
Capt USAF

AFIT/GA/EE/83D-1

Approved for Public Release; Distribution Unlimited

AFIT/GA/EE/83D-1

A Candidate Programming Language

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

by

Richard Jenninga

Capt USAF

Graduate Astronautical Engineering

December 1983

Preface

This thesis has been written to provide a template for a new effort to harness automation into productive work. Failing this, it should at least serve as a cogent argument to encourage new thinking in the field of computer science. Failing even this, it does serve to document one frustrated user's view of utopia.

The candidate language has been named D for a number of reasons. First, the successor to C (aka the next widely used language) was presaged to be either D or P by the C authors [Ritchie:2019 78]. Since C strongly influenced D, the candidate language could obtain its name for this reason alone. The ADA language also influenced the D extensions to C, in function if not form. D can be considered a functional equivalent of ADA without all the Accreted Aggrandizement. ADA without the A's is D.

The suggested method of reading this thesis is to read interesting sections first. Each section is relatively self contained. Although a complete understanding of D must be gleaned from the entire thesis there is a lot of redundance. The motivation was to permit readers who like hardware to whet their interest in that section; readers who like C to read the C comparison section first; and readers who like to see code to start with the examples. So primed, the remainder of the thesis should be more palatable.

For those who do believe that new computer architectures are required, and have a specific approach defined -- this thesis is not for you... get to work.

As with most engineering endeavors, this thesis builds extensively upon four previous bodies of work. First, and foremost, are the ideas and motivations of John Von Neumann as documented in his collected works [Von Neumann 63]. Although the proposed architecture is not what might be classified as a "Von Neumann" architecture, I submit that it is very similar to what he might have proposed to exploit the capabilities of current (1983) microelectronic fabrication capabilities. His concepts of "organs" is maintained, and extended in this age of quick and easy transplants [Von Neumann:20 63]:

> For the purposes of our discussion we shall distinguish the following organs of a digital computer: The memory, i.e. the part of the machine devoted to the storage of numerical data; the arithmetic organ, i.e. that part in which certain of the familiar processes of arithmetic are performed; the logical control, i.e. the mechanism which comprehends and causes to be performed the demands of the human operator; and the input-output organ which is the intermediary between the machine and the outside world.

Second is the work of Christopher Strachey and Dana Scott, who attempted to describe the semantics of languages, as contained in Denotational Semantics: The Scott-Strachey Approach to Programming

Language Theory [Stoy 77]. This work defines what the purpose of a programming language is, and what is important within it. While time was too short to attempt to use denotational semantics to anchor D to a solid axiomatic base, this work did motivate many of the design decisions in the hope that D might be anchorable at some future time. The most obvious feature of the language so derived is the approximate equality within the language of dynamic and passive objects.

Third is the C language. Strachey worked with several collaborators on a language called CPL, which never published or implemented [Stoy:xxiv 77]. A variant, BCPL, was implemented by Martin Richards [Richards 82] and has established itself as an important implementation language. In 1970, Ken Thompson developed and implemented B, differing from BCPL mainly in syntax because of the small size of the first B compiler (4K 18-bit words) [Ritchie:1992 78]. C evolved from B circa 1972, differing by the introduction of types, motivated by ALGOL 68 and PASCAL [Ritchie:1996 78].

Last and (perhaps not) least is the DoD sponsored Ada language development effort [DARPA 80, Ichbiah 79a, Ichbiah 79b]. The work was important in a positive sense; many important concepts were brought out of the esoteric computer science journals and introduced to language users. The DoD most certainly got an excellent engineering development version of a potentially acceptable language for their investment. However, the decision was made to standardize upon a language which many felt contained flaws [Boute 80]. One respected language designer even took his opportunity to give the ACM Turing award lecture to lambast the United States Department of Defense [Hoare:424-5 81]:

> In this last resort, I appeal to you, representatives of the programming profession in the United States, and citizens concerned with the welfare and safety of mankind: do not allow this language [Ada] in its present state to be used in applications where reliability is critical, ie, nuclear power stations, cruise missiles, early warning systems, anti-ballistic missile defense systems. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus. It may be a nuclear warhead exploding over our own cities. An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations.

Professor Hoare is now hard at work on his own language, OCCAM, in collaboration with INMOS Ltd. [Taylor 82].

In a sense, the candidate language D, is a production version of Ada. Put another way, the D architecture will support Ada programs more efficiently than current architectures.

Finally I would like to acknowledge two people who made this thesis possible. The first is Cecil Gwinn who convinced me that currently available computers were not the greatest technological innovation since sliced bread by consistently obtaining accurate results faster with his HP calculator, and some quick analysis, than I could obtain on a DEC System 10. He also introduced me to recreational mathematics, demonstrating mathematics has applications beyond terrorizing

engineering students and obfuscating intuitive results! Needless to say, I would have been unable to unlock many of the ideas in Stoy's text without his helpful insights.

The second person who should receive credit for this thesis is Harold Carter, who despite great risk agreed to advise this thesis, and who permitted me the flexibility to avert impending catastrophes.

Richard Jennings
Dayton Ohio
November, 1983

Contents

List of Figures

List of Tables

Abstract

Conventional computer architectures are obsolete. They are performance limited, unreliable and hard to program. In addition, they are able to make very inefficient use of the currently available microelectronic technology.

This state is perpetuated by the attempt to seek new languages, new operating systems, and new hardware independently; the desire to maintain compatibility with existing systems; and the desire to design with integrated circuits (VLSI) as tiny TTL. This mold is broken by the description of an architecture in which the language, software, and hardware are all designed synergistically, constrained only by the characteristics of the users of automation: people.

A candidate language is described and compared with C. Some characteristics of a program support environment are suggested. The hardware structures implied by the proposed architecture are described. Finally, two examples are provided which demonstrate the language.

While the next computer architecture to be used for 40 years is not described, enough ideas are described in detail to provide a stimulus and direction to researchers who have been convinced by contemporary computer systems that THERE HAS GOT TO BE A BETTER WAY!

I. INTRODUCTION

The Air Force, and other sophisticated users of information processing equipment, are contemplating using antediluvian architectures to meet future information processing requirements. Information processing systems are being tasked to accomplish fundamentally new and different objectives; consequently, fundamentally new and different architectures should be expected to organize these new systems.

Contemporary computer architectures are unsuitable for future information processing requirements. Simply, they do not provide sufficient structure to permit the large information processing systems to be sufficiently organized.

The role of information processors will be vastly different in the future, as will be their relationship to users. They will transition from slaves to partners. Users will evolve from programming experts. What users will evolve to is described in the next section. Past users, specifically professional programmers, possessed a relatively high degree of computer literacy, relative to the complexity of past computers. Computers essentially were, and are, used as human computers before them: to calculate results from defined algorithms.

Most future users will expect information processors to deduce theories from data. They will not have a solid notion of how information processors work in terms of fundamental physics, and of the failure modes they should be wary. The future architectures must assume responsibility to maintain the information model they present the user; during algorithm development as well as during operation.

The responsibility of information processors will also radically change in the next few years. They will regularly assume not only life

critical roles with respect to individuals, but civilization critical roles with respect to nuclear weapons and space operations; to name two applications of direct interest to the Air Force. A quote from the back of the NS 16032 microprocessor data sheet (April 1982) is illustrative:

> NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein: [life support devices and critical components are then defined]

In the past, information processors have merely been computers --- ignorant slaves dogmatically executing their instructions. No intelligent user would use computer results without independent checks as a consequence of their generally unstable behavior. This will soon be impossible for most applications, as it is already in some.

Not only will responsibility for major portions of our civilization be held by automation, but this automation will not be executing algorithms: it will be infering causal relationships from experience.

Current architectures can not be extended sufficiently. They were designed to use hardware efficiently, and issues of human comprehension and reliability were added as an afterthought. An evolutionary path to a satisfactory architecture, which can be easily understood and is reliable, necessitates the inefficiency imposed by constraints of a bygone era. Extraneous concepts must be identified and eliminated. This is the core objective of this thesis, in the context of the hardware-operating system-language model of information processing.

Hardware limitations have constrained most current architectures, and these limitations cause myriad user frustrations. The two major classes of limitations are: 1) those in which certain features,

arbitrarily chosen (at least from the perspective of a particular application), constrain capabilities and efficiencies; and 2) the arbitrary selection, and omission, of features (at least from the perspective of a particular application). An example of the first limitation is the selection of a base of the number system used by computers [Ginsberg 77]. An example of the second is the set of instructions, and register set, supported by the machine architecture. Studies are done to provide 'optimum' instruction sets, based upon programming languages, not upon what a particular user requires [Bal 82, INTEL 81, Stritter 79, Patterson 81].

An approach to this problem is work which will develop a Computer Aided Design (CAD) technique to generate hardware from the programming language definition of a program [VLSI 83, Buric 83]. This thesis proposes a comprehensive approach towards addressing the problem of achieving understandable and reliable architectures which can be efficiently designed, produced and maintained. Simply, all this thesis suggests is to bring some of the structure developed in Higher Order Languages (HOLs) into the hardware after selecting a consistent and nonredundant set of concepts.

Since hardware has been so expensive to build, extensive effort has evolved general purpose designs which can be adapted to many uses. Consequently, most people who specify actions which they would like accomplished by currently available automation can do little more with the basic hardware than warm themselves.

To permit the hardware to be used, many languages have been developed which permit the application programmers to describe algorithms in terms of a model which is closer to their application than

the underlying structure of the hardware of the machine. For examples: FORTRAN was developed to compute numerical formulas; APL was developed to support linear algebra; BLISS and C were developed to specify operating systems; PASCAL to teach structured programming; LISP was designed to manipulate symbols; and PL/1 and ADA were designed to be 'universal'. Without exception, languages used in program development systems require access to operating system and machine instructions to patch over portions of their model which is insufficient for a particular application.

As computers have become cheap and popular, hardware architectures have proliferated. System software has evolved from its past role of giving one machine many faces to giving many diverse machines one face. The UNIX operating system, which has been ported to every widely used programming environment is a example of the latter.

Conventional wisdom attempts to separate the processing environment into three areas: the hardware, the operating system, and the language. Each of these areas is currently the focus, today, of standardization. Languages are iterated to be portable across all operating systems. Operating systems are iterated to be portable across all hardware and support all languages. Hardware is designed to support many, if not all, operating systems & languages at the cost of supporting no language or no operating system well.

By developing standards at each of these levels, by utility in industry --- by regulation in the government, short term benefits are expected in addition to insight into some of the problems mentioned above. To expect short term benefits is reasonable, to expect insight into the problems just enumerated is absurd!

Suppose it is desired to define a new language with a new feature. This approach implies that the new language must maintain compatibility with current hardware and operating systems. What if these constraints cause the problems, as demonstrated by the ADA language development effort [Boute 80]?

What about enhancing an operating system? It is constrained, as are languages, by current hardware. It is also constrained by languages. If the new feature is not conceptually within the language, then the only way to exploit the feature is with system calls. Since any system supports many languages, providing appropriate system calls which do not erode the structure of the supported languages is not trivial. Even if this can be done, system to system compatibility problems avert all but the most daring programmers from using these features. The computer science literature describes, and most operating systems have available, many such system utilities. They are not commonly used because they are not integrated into the conceptual framework of neither the host operating system nor any of the popular languages.

What about the hardware? People who build the hardware do innovate! But recall that the few engineers who are capable of integrating the hardware with software into an operating system must do so before the innovations can be exploited. The language must still be penetrated.

In order for hardware to be exploited, the operating system must support access to it, or exploit it, and the programing languages must provide access to it, or exploit it. Hardware will be utilized to the degree that it is conceptually integrated into programming languages

used by those who can benefit from it. The specialized hardware must also be reliably supported by the operating system.

Special function units have been built for specialized applications, with success limited to specialized applications. But hardware has been constrained by the operating systems available, and the popular programming languages. Hardware has been further irrationally constrained by the strategy of adopting standard instruction set architectures, which although successful for IBM in the 1960's, is without any merit what so ever in the current age. Essentially, the instruction set is so far below the level of abstraction at which most programmers work, and so antagonistic to efficiently supporting the abstractions which they do use, that administrative standardization (read waste) is impeding defacto standardization at the programming language level which could, potentially, result in more efficient and reliable software at lower cost.

Initially, hardware was limited. Operating systems and programming languages were developed to ease programming of a very limited machine. The cost sunk into programs for these limited machines became great very quickly. Based upon the faulty assumption that programming could never be fundamentally simplified, it seemed a good idea to maintain compatibility to get some benefit from past programming effort. That is: maintain current (read obsolete) operating systems and languages. Although hardware is still constrained, it is by choice and need not be!

The salient question: "Is it possible to avoid the hardware/operating system/language model for programming?" The answer is an emphatic yes!, with great gains in programming efficiency, execution

efficiency and collateral reliability. These gains are the consequence of unifying the information processing system model so the language, global architecture (operating system), and local structure (hardware) are but three instantiations of the same monolithic conceptual entity.

There seems to be substantial agreement that information processors will evolve; the major questions to be answered are: when? and how?

This thesis attempts to answer these questions respectively, now, and by reconstructing information processing architectures from first principles to be compatible with sophisticated users while exploiting current and projected hardware fabrication techniques.

The following thesis is built around a language definition, and development of a complementary global and local architecture to support it. A comparison will be accomplished to show that the resulting system is at least as competent as the C programming environment. An overview of the thesis organization, by section title, follows.

Architectural Objectives defines the term architecture in the context of this thesis. Characteristics of the users of the architecture are developed, and then the architecture is summarized from their perspective. The section concludes with identifying the constraints imposed upon the architecture.

Approach to Algorithm Specification describes the semantic concepts implemented within the language. Using as a point of departure the sophisticated user, the fundamental ideas of the language are developed. This section describes the ideas of the language unburdened by syntax.

An Introductory Example: Rational Arithmetic provides a taste of the language before the syntax has been introduced. The example provided is simple enough to permit the style of the language to be

demonstrated and understood before the language has been studied in detail.

Language Description focuses upon the syntax of the language. This section serves as a preliminary language reference manual.

Algorithm Example: A Vector Dot Product demonstrates two of the more powerful features of the language, generic algorithm definition and parallelism, in a straight forward example.

Programming Support Requirements first describes the three representations through which algorithms metamorphose during their life from user definition through machine execution. Programming development aids are discussed in a speculative, as opposed to definitive, context.

Hardware Requirements describes particular architectural features which have been developed in concert with the language, and are consequently required to support the language efficiently.

Language Analysis provides an informal argument that the proposed environment is sufficient to replace existing programming environments, and necessary to instantiate future proposed systems. Since much of the motivation for this language came from C, the informal argument takes the form of an approximate comparison of the two languages arranged in the format of the C reference manual [Kernigan 78].

Conclusions summarizes what should be done to develop a working model of the architecture, suggests work which should be accomplished to improve the architecture, and states why the proposed architecture development should be continued in light of current language and hardware development efforts.

The fundamental objective of this thesis is to stimulate serious thought about restructuring the information processing model by

providing a rough heuristic description of an internally consistent architecture with apparent potential for realization after refinement.

II. ARCHITECTURAL OBJECTIVES

This section first defines the term architecture, and then motivates the development of a new computer architecture. Users, introduced but not yet described, are characterized and the users' perspective of the architecture is developed. Limitations which constrain the architecture are mentioned. This section closes with a summary in which the priorities of the architecture are emphasized.

The focus of this thesis is to define a nutritive design environment that is realizable; hence initial emphasis will be placed upon the motivation for and definition of such an environment. The following sections describe realization issues.

Motivation

The term 'architecture' has been subject to many interpretations which have specific meaning within the hardware, operating system, and language model of information processing. To communicate with the word outside of this model, a definition is required:

> The art or science of building; specifically: the art or practise of designing and building structures, especially habitable structures, in accordance with principles determined by aesthetic and practical or material considerations

in this case taken from Webster's Third New International Dictionary (1966).

In the context of information processing systems, building refers to engineering an information processing system. Structures refer to flexible fundamental building blocks organized within a language. Habitable means pleasant to use, as a dwelling would be pleasant to live

in. Aesthetic implies that the information processing system architecture captures users thoughts unadulterated: that is, the beauty of an algorithm should not be corrupted by the limited language which must encode it. Practical insures that new architectures must always strive to improve upon current methods; uniqueness is not sufficient. Material, as opposed to ethereal, considerations insure that an architecture is an aid to suggest how information can be organized, as opposed to how one could conceive of information being organized.

To summarize, an information processing architecture should serve to structure automation to be user habitable. This thesis is but one attempt to improve upon the computer architectures which are commercially available.

Automation is not an end in itself; consequently its purpose must be reflected in the architecture. The purpose of automation, as with all tools, is to aid and extend user capabilities. This can be done in essentially two ways. As with all tools, automation contributes to sensation, actuation or both. Automation can provide control and information bandwidth reduction. Two forms of bandwidth reduction are possible: classification (or selection), and concentration. Concentration of information can take the form of deduction or induction. Deduction implies that general rules are applied to specific cases to determine actions, eliminating the need to store specific responses for each case. Induction implies that specific cases are used to formulate general rules, eliminating the need to store minutia associated with each case.

Given this sketch of what automation can accomplish, a test for the utility of specific automation can be defined. Consider a task within

the framework defined above, and quantitatively estimate effort an unaided user would expend. Divide this by the equivalent estimated effort a user would expend assisted by automation; creating and validating the task specification.

$$\text{UTILITY(user, task)} = \frac{\text{unaided user effort}}{\text{automation aided user effort}}$$

This function, of the task and the user, provides an indication of the suitability of the automation available. For example, a utility of about 1 or less would indicate that the automation available should be avoided.

The goal of an information processing system architecture is to maximize the utility function defined above over a set of users and a set of tasks. By carefully selecting user characteristics, and coupling the architecture tightly to them, the utility function, applied over the domain of qualifying users, can be increased.

User Characterization

The user characterization is critical and will shape the architecture. The user has responsibility for instantiating automation concepts. As such, two directions are possible: to structure the architecture to support the requirements of dedicated system programmers, or to strive to adapt it to professionals in other professions so they may use it directly.

System programmers currently provide, in theory, a friendly interface between automation and professionals. They realize system capability, and can perform limited debugging. Limited, because

algorithms are usually not independent of the automation. Programmers exist to increase the utility function of the professionals, but in many cases must function as a professional in the field they are supporting. Invoking the suggestion that automation does not exist for its own sake, it appears that if possible the professionals should be supported directly.

To target professionals directly, it is important to notice their relative strengths and weaknesses juxtaposed those of programmers. A dichotomy between syntax and semantics is apparent. Programmers can manage multiple complex syntaxes, with fixed semantics... to wit commonly used languages and operating systems. Professionals require a simple syntax, since its mastery requires extraprofessional study. However, they can easily handle the extensible and powerful semantics within their professions.

Mathematics provides precedence here. It provides structured extensibility from a common basis. It provides a compact notation which is rich in semantics with a simple syntax. It can be tailored to a target group with precision. It encourages researchers to be mutually supportive by catalyzing efficient communication. It permits results to be exploited by applying a utility test, recognizing the inherent limitations of an algorithm. Finally, it is decoupled from material implementation; it is abstractly selfconsistent and complete.

The "user" is now chosen to be a skilled professional, without an extensive education in computer science. The language syntax must be necessarily small and logically extensible. The semantics must be powerful, and consist of fundamental building blocks to permit language extensions to benefit from structure. The architecture must support the user in an internally consistent interactive environment.

The impedance between the users' comprehension of an algorithm specification and its realization by automation must be minimized. Just as optimal power is delivered to a load when the impedances are matched, the information bandwidth each way, between the user and the algorithm development aids, should be equal and maximized. High information bandwidths imply shared contexts, which suggests that the users model of the automation environment should be implemented directly.

Put precisely, between the language employed by the user and the machine code interpreted by the hardware there should exist a bijective, or one to one and onto, transformation.

The architecture has been justified by the desire to support a sophisticated user. The user's perspective will now be used to introduce the architecture.

User Perspective

In order to provide a useful overview of the architecture, two views are important. The first is the algorithm specification process; how does the user transition algorithms to automation? The second is the algorithm life cycle; how is the algorithm born, how is it implemented, how is its performance improved, how is it maintained, and how is the algorithm finally replaced.

The following overview starts by describing the algorithm specification process in terms of the architecture information processing model.

The information processing model embodied by the proposed architecture consists of four basic ideas. They are definition, declaration, action, and context. First, objects must be defined. Second, they must be declared, or instantiated. Third, they may either

act upon other objects or be acted upon themselves, and fourth, the characteristics of an object are not entirely described by its definition, but are affected by other objects within its environment.

The user first characterizes the objects which will constitute the automation environment. These objects can be either static or active. The term 'user,' in this context, is broadened to include a specific professional field.

Static objects can be either scalar, which denote elements of the set of discrete real numbers, or structured, which are composites of structured and scalar objects and denote n-dimensional elements in an n-space. A house number might be a scalar object, while an entire street address constituted by semantic units is a structured object. Active objects are either operators or functions. Integer operators are either monadic or binary, and return an integer. This concept is generalized to objects. Functions require optionally one object and return optionally one object. Functions do not inherit the visibility of the environment from which they are called, but may be passed portions of it via an argument (by value) and by a concept called linkage (by reference). Implicit side effects within the calling function do not occur.

A definition is a description of an object, just as this thesis describes an architecture. A description must be declared, or instantiated, or realized, or built before it can do anything. A declaration creates an object from a definition and names it. It is then real and can be used. For example, a subroutine is defined which computes the trigonometric sin function. It is declared with the name "sin", and "sin x" has the conventional meaning.

Actors are processes which act upon static objects. The user specifies these operations on scalars and structures by operators and functions he has defined. The two fundamental actions are evaluating expressions and assigning values to named objects. In addition the language supports intertask communication, iteration, conditional execution, and dynamic object allocation. The user's major challenge is decomposing his algorithm into sequential tasks which minimize intertask communication and overall elapsed execution time while maximizing opportunities for parallel execution.

The context in which an object is declared, and used, is important. In the "sin x" example above, the definition of x was not explicitly identified. It could have been defined when sin was declared, but in most cases would be deferred. The trigonometric sin can be defined in terms of other objects: the operators addition (+), subtraction (-), multiplication (*), and division (/). Multiplication and division can also be defined in terms of the operations addition and subtraction, but for efficiency reasons multiplication and division are often implemented directly in hardware. In theory, as in this architecture, the semantics of sin are dependent upon these constituent operations. The linkage between these ideas occurs, in contemporary parlance, at 'runtime' without performance penalty because of architectural innovations (see section V for details).

The next section, "Approach to Algorithm Specification", continues the development of these ideas. The remaining part of this section will describe the life cycle of an algorithm.

An algorithm is born when it is fully described and communicated. The language should, and the proposed language does, efficiently share

structure with the user. It accomplishes this sharing by adopting the organizational perspective of the user with respect to objects. Except for the basic building blocks consisting of the means to construct scalars, structures, functions and operators, all of the organization within the language is user defined. A language definition language is being proffered.

This language is specifically designed to flexibly accommodate the user. Consequently the specification and debugging of a working algorithm should be the simplest method to communicate its accurate description. Algorithm specification is catalyzed by an interpretive environment, with a semantically oriented program editor and execution monitor (see section VII for further discussions about programming support).

Once the algorithm has be specified, it must be implemented. The distinction is that an implementation is constrained by operational limitations, such as a time line, or execution costs. The algorithm must be decomposed into concurrently executable sequential tasks with the aid of tools to interactively monitor intertask communication and interactively monitor task execution.

Restructuring the algorithm may not be sufficient to meet operational limitations imposed upon the algorithm, and exotic technologies may be required. Computationally intensive portions of actors can be isolated, and compiled into hardware; the compilation process is quite simple compared to that required to instantiate the average software subroutine in hardware. A conceptually similar method has already been described by Buric [Buric 83]. The architecture supports a linkage to active objects which is independent of their

implementation in either hardware, software or both. In all cases, task control is maintained by the standard kernel which is the machine language interpreter. This will increase performance, reliability, and compatibility between hardware, users and algorithms; in that order.

Maintenance can be done modularly. Suppose at some point, returning to the sin example, it is desired to replace a Taylor series expansion with a Chebychev series expansion. It can be done once for all objects for which sin has meaning.

This can be done because sin contributes a formula, in the form of a Taylor's series or Chebychev expansion, in terms of the basic arithmetic operators '-', '+', '*', and '/'. For any domain, represented by x, for which these operators have meaning... so does sin x. This exemplifies the importance placed upon context by the architecture.

The concept of system replacement will be itself replaced by the practice of incrementally changing the hardware or the algorithms. Since the algorithms are not tightly constrained by their automating system, incremental modifications will be less constrained, hence cheaper, hence used more often to meet system capability shortfalls: instead of replacing the entire information processing system. Algorithms, since they are represented in a language tailored to the field from which they sprang, will be easy to understand and reliably modify: that is change or correct without introducing new bugs or design errors into the system.

Implementation Constraints

Any architecture, because of practical and material considerations, is constrained. This architecture is primarily constrained by the

people who constitute users. It must reflect a human comprehensible language, which implies what might be considered inefficient structure and hierarchies. It must bend the otherwise ideal machine architecture to permit educable users to think within the structure it provides. That is, not ones and zeros: 10010100 10101101 10010011 00011101. (The so motivated reader may wish to compare the meaning of these four bytes in the several machine languages in which he is personally intimate).

In the short term, it appears that VLSI based upon solid state physics, specifically semiconductors, will be the implementation technology of choice with the attendant interconnection limitations. However, organic molecules offer a light on the horizon promising density, speed, connectability and selfrepair which cannot be ignored [Ostroff 83, Barker 80]. How many presaged VLSI in 1950?

Priorities

In the remaining portion of the thesis, as the topics gradually become more applied, it is important not to lose sight of the primary motivations. These are now summarized.

The interface to the user must be optimized. The user must not be bored or frustrated. The impedance between the user and the automation must be minimized. This implies two way communication. The purpose of the language is to facilitate efficient two way communication, which implies a bijective transformation between the language and the machine instructions. Communication implies shared context, and to maximize the context shared, the context should be dynamic and extensible. The user must be able to easily raise the level of abstraction without sacrificing architectural structure: structured extensibility is required. The language must embody a concise system model to

unambiguously (over the carefully defined user population) interpret a specification, and the architecture must ensure that automation behavior consistent with the system model is presented to the user.

Finally, the architecture must support semantic structures convenient to people efficiently, directly, and comprehensibly. In the next section these structures will be described.

III. APPROACH TO ALGORITHM SPECIFICATION

The conceptual characteristics of D will be informally introduced in this section, before the syntax is described. The following panoramic view is designed to aid organization of the language details described in section V, entitled "Language Description".

The most difficult part of designing any system is partitioning the system into constituent, compatible, comprehensible subsystems. The structure of this programming language flexes to meet application requirements. It does not provide a canned solution to any particular application but instead provides a nourishing environment for many. In a sense it is a "language definition language", and "application programs" are really specialized languages catalyzing communication between man and machine. By attributing characteristics to the architecture which support experts familiar with it, and their field, novices are sure to be well supported.

As will be seen, the language is interpretive, structured, and extensible. The user is guided by the environment in the creation of a robust, efficient and extensible program.

Objects are fundamental to the language semantics. They not only hold values but may connote meaning. Denotational objects are considered passive, and are called either "scalars" or "structures". Connotational objects are considered active, and are called "actors". For example, "operators" and "functions" are actors.

Certain basic objects are defined within the language. These must be used as a basis for object definition and declaration. An application program must define classes of objects, declare

instantiations of defined objects, and apply actors to objects within the language structure to yield program results. The language consists of structure to:

* dynamically create objects,
* control statement sequencing with iteration and conditional statement execution,
* communicate between actors, and
* precisely control the scope of object names.

Section IV contains an limited example of this structure in the form of an algorithm to add rational numbers. Although the syntax will not have been introduced, the example demonstrates how portions of the language work together. After this example, the syntax is described in section V. It, in turn, is followed by an more substantive program example which demonstrates generic algorithm specification and parallel execution.

Introduction to Objects

In the course of specifying an algorithm, classes of objects must be specified. An object is something that is capable of storing or manipulating a value. Variables, constants, files, functions and operators are all objects. A name is not an object.

Characteristics common to a set of objects are abstracted and bound into a defined class. An object is created, or declared, as a member of a class. Each object shares its characteristics with other members of its class with the exception of its name and its value. A unique name is given to each object when it is declared. Its value need not be unique.

'Value' is used in both the denotative and connotative sense. Objects can be either static (passive), or dynamic (active). A passive object is acted upon, while the active object does the acting. For example, algebraically the letter 'x' may name, or denote, a real number. The function sin x denotes a real number as well if x is known, but also connotes trigonometric theory whether x is known or not.

There are four types of files which factor the characteristics of objects permitting efficient and modular algorithm automation. Characteristics of static objects are described as scalars and composites of scalars, called structures, in state definition files. Characteristics of dynamic objects are described as operators and functions in terms of sequences of transformations upon static objects in actor definition files. The values of objects are declared and maintained in storage files.

Context, which controls the interpretation of object characteristics, can be organized using bundle files each consisting of a list of file names.

Introduction to the Programming Method

Every language has a programming method requisite to its effective use. D is no different. Although programming is recognized to be an iterative process, a nominal sequence of steps can be identified.

First, templates must be defined for each class of static object which is required to maintain values by the algorithm.

Second, the transformations (or actors) must be defined. Each actor argument class and result class, if any, must be determined and declared. Objects referenced by the defined actor must be determined and realized. Any transformation must be defined in terms of the

transformations available in either preexisting definitions or hardware. Each of these transformations, implemented as expressions, must be integrated into the target algorithm via control: conditional expression evaluation and iteration.

Third, defined objects must be declared or created. Active objects specialize the class of their arguments, while passive objects specialize the class of the value they maintain.

Finally, the juxtaposition of generic class-less actors, class definitions, and object declarations permit algorithms to be defined and manipulated efficiently at user created levels of abstraction.

Basic Set of Predefinitions

The proposed architecture and language attempt to provide a structured and extensible environment for users to adapt automation to their requirements efficiently and reliably. A core, from which everything can be built upon, is predefined within the language.

The architecture understands a practical subset of discrete real numbers, and their associated arithmetic and logical operators. Predefined arithmetic operators are (=, +, -). Note that (*, /) can be defined in terms of (+) as can the logical operators. Fixed, float, character, and other types which often are considered part of contemporary languages are all user defined.

All the classes and their supporting operations and functions must be defined in terms of the predefined operations. This does not imply poor performance because of the architecture, as will be seen, is designed to support hardware instantiation of compute bound actors. A "predefined" set for any realization need not be the minimal "predefined" set of the architecture, but should include it.

Four Important Files

The architecture is constructed to support four basic concepts. These concepts are combined to automate verifiable and reconfigurable algorithms.

The structure of a file contains four entites with specific functions. They are the file class (either state, actor, storage or bundle), the name of the file, the linkage section (identifying other files to be shared or copied to correctly interpret the following body), and the body.

Files can be either copied for exclusive use of the body, or they may be shared. Usually definition files are shared, while declaration files are copied. If two concurrent actors need to share access to objects, then they might share the storage files instantiating the shared objects. The linkage section also permits objects to be renamed for user clarity and machine efficiency. The differentiating portion of each file type is its body. Each of the body functions is introduced in the following paragraphs.

The first two classes of files (state and actor) are required to construct the basic building blocks of an automated algorithm. Scalars and structures are defined by state definition files. A scalar is an object with one value. A structure is a composite of scalars, other structures, and pointers to active objects.

Actors, operators and functions, are defined by the actor definition file. Operators are named by special operator symbols and require one, optionally two, arguments and always return a result. Functions are referenced with object names and require, optionally, one argument returning, optionally, one result. Since structures may be arguments, this is not as restrictive as it may first appear.

Once building blocks have been designed, they must be constructed out of compatible "materials". In this architecture, this is analogous to creating objects, both active and static, with compatible classes. Storage files are used to declare objects, or create entities with names and values from the definitions.

Bundle files organize the logistics of integrating the building blocks into an automated algorithm, or application program. Related state files, actor files, storage files, and bundle files can be bound together into a structured unit constituting a component in an even larger system. This file class effects the agglomeration of related files into an entity which can be manipulated as a unit. It provides an amenity to encourage modularity and functional factorization of definitions.

In the following subsections, each of the file bodies will be decomposed into its salient concepts. The intent is to motivate the capability included in the language. Precise specification of the language is deferred until section V.

Definitions

In order to conceptually manipulate many objects, it is imperative to factor the characteristics shared by sets of objects into classes. During algorithm specification this process has already been largely accomplished motivated by analysis as opposed to algorithm automation issues. The problem is to construct blueprints of the many objects which must be modeled by the automation using as many common building blocks as possible. This must be done without constraining analysis or obfuscating the algorithm in its executable form.

The objective is to concentrate shared information into single files. This information can be efficiently referenced using linkage without respecification. Minor changes should be localized to one small file. This is an example of the orthogonality the architecture is designed to support.

Each state definition must have a unique class name. Each state definition defines a machine representation and a user representation. Definitions serve to communicate the users design to the automation in three major areas.

In the first area, scalar representation, the range of values must defined as well as the compatibility of these values with other classes. All scalar values are represented as a countable set, bijectively mapped onto a set of discrete whole numbers. Through this discrete set of whole numbers, the predefined operators are given meaning.

In the second area, multiple scalars and known structures are organized into new structures. The machine format is specified in terms of scalars and known structures. The user format is specified in terms of scalars, known structures and quoted literals, with the capability to specify whether leading or trailing zeros in the representation of a value are significant. For example, in a floating point representation trailing zeros are not significant in the mantissa. Zeros on either side of the decimal point are; and so on. The important point to recall is that the user format is bound to the class of the object. It is not determined by an input or output actor.

In the third area actors, which manipulate static objects, are defined as sequential procedures. The body of an actor definition consists of an operator or function declaration which names the function

about to be defined, and declares its arguments. Two types of dynamic objects can be defined: functions and operators. Operators require at least one argument, and may be either prefix, infix, or postfix. An operator must return one value. The operator name space and all other name spaces are disjoint.

Functions may optionally have one argument, and may optionally return one value. They are named from the same name space as static objects.

It is possible to defer the identification of the class of an argument, or a returned value, until actor declaration or even execution. The argument of an operator or function can use its class, if defined, as a parameter. This can be handled in two ways: first, the class can determine which actor is selected; second, the actor can be defined in terms of other actors which are visible during execution. In the latter case the function is class independent.

Actors can perform conversions from one class to another, but they cannot accept arguments of different classes. However, if a function is invoked with arguments of two different classes and a functional is available in the execution environment to perform the requisite class conversion it will be invoked to perform the conversion automatically.

Each actor definition contains a block which sequences object manipulations invoked during execution. Blocks will subsequently discussed.

Declaration

After all the classes relevant to the algorithm have been defined, the blueprints for the building blocks are complete, and the building blocks are ready for construction. The purpose of a declaration is to

instantiate an object and then bind it to a referencing method and a class definition.

The class definition serves as a template for the declared object. There are five kinds of templates: scalar, structure, operator, function and file. The template provides a class name denoting abstract properties of an object. It also provides information to permit access to components of a structured object. The range of an object is defined by its declaration template. For actors, the domain of the arguments and the range of the result are defined.

Four basic referencing methods are possible in the language. A tree model underlies structured classes, and this method is controlled by the definition of a class. The other three methods are arbitrated by the object's declaration. A direct referencing method requires a name to refer to an object. An indirect reference permits a name to point to another name which references an object. Objects so declared may be initialized to known names not values. A group of components may be collected together into an array, and referenced with the aid of an index. The defined class of an object and its referencing technique may be abstracted and used as a template for yet another object. In such a manner, pointers to pointers and multidimensional arrays may be declared although a series of steps is required.

Objects may be created statically by creating and copying storage files. They may also be created dynamically with the new keyword and a known pointer object. The dynamically created object is appended to the pointer's owning storage file. The object reference value is assigned to the pointer, and the object may be initialized when it is created.

The life of object begins when it is instantiated by a storage file or when it is copied as part of a storage file, or when it is declared by a new statement. The life of an object ends when it is no longer referenced by either the declaring actor, or any subsequent referencing object. The keyword null can be used to dereference objects from pointers.

Linkage may be used to locally introduce new objects with known names. The old objects are hidden only until the end of the block in which the new objects were introduced.

Action

Action occurs when a new value is assigned to an object. Three ingredients are required:

- the logical name of the object to be assigned the new value,
- an expression which will evaluate to the new value, and
- an assignment operator which effects the assignment.

First the concept of a logical name will be discussed; second, expressions will introduced with the additional complications to action they engender; and third assignment will be described. For now consider the action connoted by an actor to be defined by a list of statements assigning expression values to logical names and executed sequentially.

The simplest form of a logical name is an object name which provides a direct reference to the object of interest. Some objects, called pointers, hold, as values, the logical names of other objects. An ambiguity between naming the object holding an object name and an object holding a object value is resolved by preceding the name with a

special symbol, '#', when the value holding object is required. That is if "ptr" is a pointer name, "ptr" refers to an object name while "#ptr" refers to an object value; the object value of the object name pointed to by "ptr".

File names are also considered to be logical names, but of a particular class. Each logical name is bound to a particular class. Objects referenced by a pointer must all be of the same class. No object can be declared to be of a generic class which permits it to maintain values of all classes.

A logical name can also refer to components within a structure. The structure must be named, and then following a special symbol, "'", the component must be named.

Many objects derived from a class may be bound into a multidimensional array. The entire array, a dimension, or a component may be referenced using a logical name consisting of an array name followed by indexes in brackets, '[' and ']', which select dimensions and components.

Pointer names and array names are logical names themselves, permitting nested indirect references. A renaming capability within the linkage model permits logical names to be streamlined.

Assignment is the one operation in the language which is predefined for all objects regardless of class. The class of the expression on the right of the assignment operator is coerced, if required and possible, to match the class of the logically named object on the left of the assignment operator. Assignment occurs only if the class of the logical name, on the left, is identical to the class of the value, on the right. The resulting class required for assignment does influence the selection

of operators within an expression.

Careless definition of operators can lead to ambiguous expressions with values which are not precisely defined. The language in no way supports such careless programmers. A linkage facility is provided which permits competent programmers to precisely tailor the scope of actors to insure that this problem is avoided.

The simplest expression consists of a logical name. The value of such an expression is simply the value of the object referenced by the logical name. Normally, an expression consists of multiple logical names separated by functions and operators with various levels of precedence. Operator precedence can be explicitly controlled by parentheses. Each function and each operator is defined in terms of a sequence of statements within an actor definition.

An actor may be evaluated by hardware, another task, or by a declared function. The context of a declared function invocation is that of the calling environment. If control is passed to another task (active actor) which fields the function call with an accept, only the argument values are passed. The execution context of the fielding task is used to evaluate the function. This is also true if hardware is invoked to perform the evaluation.

The concept of an expression has been discussed, as has the concept of assignment permitting connotative results to be statically maintained. An algorithm generally consists of multiple expressions, bound together with control structures to implement iteration and conditional execution. In the case of this language, dynamic variable declaration and local scope control are also supported to simplify the definition of actors.

The statement is the basic unit of sequential execution in D. There are four statement types, three of which are expressions. The aforementioned expression is syntactically a statement. The return statement is an expression which consists of the language defined function "return" which may have an optional argument. This function passes control back to the calling function. The new statement is is an expression which consists of the language defined function "new" which requires the logical name of a pointer, and optionally an initialization expression.

Since D is a block structured language, unlike any widely used language, it is not surprising that the statement which is not an expression is a block. Each actor is functionally defined by a block. A block essentially is the one and only control structure in the language.

Foremost, a block is a list of statements to be executed in turn. Each block begins with linkage making the block the fundamental unit of scope control within the language. Since the list of statements within a block may consist of other blocks, "block structure" as used within other contemporary languages is descriptive here.

However, D blocks support conditional execution. Each statement may be prefaced by a guard. The guards of all the statements of a block form a vector of expressions which is evaluated each time the block is entered. The resulting values of the guard determine whether the guard is open or not. Statements without guards are always considered open.

Upon entering a block, control passes to the first statement which has an open guard. In addition to a guard preceding a statement, a continuation follows it. Depending upon the continuation selected,

control may pass either to the next open statement, or to the next statement regardless of the guard value. The "next statement" may be put into the context of any of the enclosing blocks, assuming that the enclosing blocks are labeled. Each block may be preceded by a header containing a block label.

In addition, D blocks support iteration. Continuation symbols are provided to cause control to reenter a block, or to terminate it.

In addition, D blocks provide real time support. In addition to labeling blocks, the header contains an expression which defines the value of open guards, and a switch which enables real time support within the block. When enabled, control within a block passes through the first guard to open, as opposed to the first open guard in the list after all previous guards have been evaluated and are not open.

Context

The intent of the language is to be tailorable to abstractions employed by users in their endeavors. Context permits ideas and connotations to be factored into their constituent parts, and referenced precisely. Each idea can become a building block of a larger abstraction. Key objectives are modularity and controlled communication between the modules, where 'module' is used interchangably with file.

Communication permits the plethora of factored parts to be integrated into an algorithm. The architecture supports local communication, defined to be interaction between related parts of an algorithm executing simultaneously in one processing system. The architecture also supports global communication, defined to be interaction between different algorithms which execute at different times on different processing systems. Both forms of communication are

implemented consistently with respect to each other.

Communication is implemented via linkage. All object names for scalars, structures, files, operators and functions have no meaning unless the module in which they occur either defines them in terms of known quantities or accesses, via linkage, a module where they are defined, or declared. By changing links, a similar idea can be exploited in vastly different contexts.

Modules may be shared or copied. If modules exist then they may be accessed by establishing a link to them. They may be shared with other using processes, or copied to create a private instantiation for exclusive use. Definition modules would normally be shared because they are not modified by the modules establishing links to them. Storage modules would normally be copied because they are normally modified by using modules. Most algorithms do not expect static objects to change their values without being operated upon by the algorithm. However, one method of establishing communication is through shared storage modules.

Once linkage to a module has been established, the object names defined or declared within the module are accessible. If name collisions occur, or if the names within the module are inappropriate, the object may be renamed. In cases where a reference consists of several levels of indirection, renaming provides a direct name for the object.

Bundle files hierarchically organize modules which are required to instantiate an algorithm. By linking to a bundle file, all of its constituent files become accessible. Modules may be organized as libraries of functionally related operators as opposed to algorithms.

Context manipulation permits connotations to be factored from their

implementation, as they are perceived. The connotation of the trigonometric sin function is independent of the numerical technique used to implement it. Similarly, the connotation of a Chebychev approximation is independent of the function approximated. Through the context management provided by linkage, these two ideas can work together, and with many others, without a great deal of redundant implementation. Operator structures as well as data structures may be extensibly supported.

IV. An Introductory Example: Rational Arithmetic.

Although the following example is mundane, it demonstrates how parts of the language work together. The algorithm takes several rational numbers of the form "x/y", such that 'x' and 'y' are integers, and adds them. The sum is presented in relatively prime rational form.

The algorithm consists of several steps:

* converting the entered expression into internal form;
* performing addition of the rational numbers;
* eliminating factors common to the numerator and denominator; and
* presenting the final result.

Two classes need to be defined, one to specify rational numbers and one to specify addition over rational numbers. An expression is entered on a keyboard as it might appear in a program. The result is evaluated and displayed on the terminal screen. In this case the algorithm is defined to be a function named "add". Assume that two user written functions, "get" and "put", are available. Assume that "get" obtains input from the user, and that "put" displays output to the user. These functions have access to the definitions about to be described. Also assume that the file "standard definitions" contains definitions of the operators "<=" less-than-or-equal, "==" is-equivalent-to, and "&&" logical and.

When code is given as an example, as in the remainder of this section, a general description of the chunk of code will be provided first. Then the code will be listed on numbered lines. The numbers, not part of the language, serve to tie the following more detailed explanation to the code presented.

First, the definition of a rational number must be specified. It consists of two integers, separated by a '/', and leading zeros (those zeros on the left of the integer) are not significant and are not printed. Since rational numbers consist of integers, access to the standard definitions are needed. A period ends the definition file.

Note that an ambiguity is introduced with respect to a rational number of the form "a/b" and the integer division operator '/' since both 'a' and 'b' are integers. This is resolved by applying resolving operator names <u>after</u> all passive objects have been recognized. For example if the result of an expression is assigned to a integer object, then "a/b" is not interpreted to be a rational number, for 'a' and 'b' integer.

*** code segment ***

```
1: state rational_number_definition
2: share standard_definitions    -- where integer is defined
3: rational :: {    )                -- ) means 02   = 2
4:                  integer: numerator
5:                  /)               -- ) means 2/09 = 2/9
6:                  integer: denominator
7:             }
8: .
```

*** explanation **

1: The file type is a "state" file, and the file name is "rational_number_definition".
2: The scope is opened to include the file named "standard_definitions".
3: A structure with the name "rational" is declared, and the leading zeros of the component declared on the next line are suppressed when they are displayed.
4: The first component is declared to be of class integer, a predefined class, with the name "numerator".
5: The separator between the two components is declared to be a '/', and as in (3:), the leading zeros of the next component are to be suppressed in the user format.
6: Another integer is declared with the name "denominator".
7: The structure declaration is terminated.
8: The state file is terminated.

The '=' operator is always defined: it always copies, verbatim, the contents of one object to another, checking to ensure classes involved are compatible. Other functions and operators are required, and must be defined.

As before, the standard defintions are required since integer operations are used to define the operations on rationals. A declaration file is also required, although it has not yet been defined. This is permissible, since linkage to these files is required only during the execution of the functions defined here, and linkage occurs dynamically just before execution.

*** code segment ***

```
1: actor rational_actor_definitions
2: share standard_definitions
3: share rational_number_declarations
```

*** explanation ***

```
1: The file is of type "actor", and named
   "rational_actor_definitions".
2: Scope is opened to include the file "standard_definitions" and
3: "rational_number_declarations".
```

The first function defined takes a rational argument and provides a rational result. Since the structural definition of rational numbers is verbose, it is convenient to give each component of the rational number operated upon another more succinct name. With brute force 'i', declared in "rational_number_declarations" along with all the other non-standard actors and structures, is used to attempt a reduction of the rational number argument. It is initialized to 1, and then incremented until it is greater than one half the denominator.

If it divides both the numerator and denominator evenly, then the numerator and denominator are reduced. The index 'i' is incremented,

and this iteration is continued until 'i' is too large.

*** code segment ***

```
 1: reduce :: in  rational: r
 2:           out rational: r
 3:         rename r'numerator N
 4:         rename r'denominator D
 5:         { i = 2;
 6:           new_divisor:
 7:           { i <= D/2,
 8:                $ factor:
 9:                  {   ((D/i) * i - D == 0),
10:                   && ((N/i) * i - N == 0),
11:                    $ { D = D/i
12:                        N = N/i
13:                        } factor \
14:                      i = i + 1 new_divisor \
15:                      }
16:             }
17:           }
```

*** explanation ***

1: The defined function is named "reduce", and its formal argument is of class "rational", and is named 'r'.
2: The range of this function is also the class "rational", and the formal name of the result is 'r'.
3: The component of 'r' named "numerator" is renamed 'N'.
4: The component of 'r' named "denominator" is renamed 'D'.
5: The block defining the function "reduce" is opened, and the scratch variable 'i' is initialized to 2.
6: The next statement is a block, and this line labels it with the name "new_divisor".
7: The block is opened, and the expression "i <= D/2" is evaluated. Since the line ends with a comma, the logical line continues to the next physical line.
8: The '$' symbol indicates that the preceding expression is a guard. In this case if the preceding expression evaluates to a positive nonzero value, then control passes to the following statement (9:), otherwise control passes to statement (16:). The following statement is a block, and is labeled "factor".
9: The block is opened, and D is tested for divisibility by i, the comma terminating the line signifying that the logical line continues.
10: And ("&&") N is also tested for divisibility by i, and the line continues.
11: If both D and N are divisible by i then the following unlabled block is entered, else control passes to (14:). An i is factored from D.
12: An i is factored from N.
13: The current block is closed, and control passes to the beginning of the block labeled "factor".

```
14: The value of i is incremented, and control is passed to the
    block labeled "new_divisor".
15: The current block is closed. (factor)
16: The current block is closed. (new_divisor)
17: The current block is closed. (reduce)
```

Now the definition of rational number addition will be accomplished. Two rational arguments result in a rational sum, and the rational sum is obtained by component by component addition after giving each term a common denominator. Since the reduce function is available, the sum is reduced before it is returned.

*** code segment ***

```
1: + ::   in  rational: ad1, ad2
2:        out rational: sum
3:           {
4:           sum'denominator = ad1'denominator * ad2'denominator
5:           sum'numerator = ad1'numerator * ad2'denominator,
6:                         + ad2'numerator * ad1'denominator
7:           sum = reduce sum
8:           }
```

*** explanation ***

```
1: An operator with the name '+' is defined which requires two
   rational arguments. The formal names are "ad1" and "ad2".
2: The result will be rational as well, with the formal name
   "sum".
3: The defining block is opened.
4: The component named "denominator" of "sum" is computed,
5: and the component named "numerator" is computed,
6: over two lines, using the predefined integer operators.
7: The result "sum" is reduced to lowest common denominator form.
8: The defining block is closed, and sum is shipped off to the
   calling function.
```

For completeness, the rational subtraction operator is defined without explanation.

*** code segment ***

```
1: - ::   in  rational: sb1, sb2
2:        out rational: sum
3:           {
4:           sum'denominator = sb1'denominator * sb2'denominator
5:           sum'numerator = sb1'numerator * sb2'denominator,
6:                         - sb2'numerator * sb1'denominator
7:           sum = reduce sum
8:           }
```

Partly to demonstrate the language, the C operator '+=' is defined. It is defined in terms of the '=' operator, which is universal, and the '+' operator which must be available at runtime.

*** code segment ***

```
1: += ::   in  $sum: sum, inc
2:         out $sum: sum
3:            {
4:            sum = sum + inc
5:            }
```

*** explanation ***

1: A function named "+=" is defined which has formal arguments of any class with two formal arguments named "sum" and "inc".
2: The result has the same class of the arguments, and is one of the arguments: "sum".
3: The defining block is opened.
4: A new value for sum is defined using a presumably visible operator named "+" capable of operating on objects of the class of "sum".
5: The defining block is closed.

Now the object function can be defined. It does not take any arguments per se, since it invokes the input function "get" and output function "put". If "add" was omitted from the command line below, the result would be nearly the same. The error handling might be different.

When "get" finds a rational, or '+', waiting in the input buffer, it returns a 1 or a '+' respectively. It assigns the value of the waiting rational to the object declared as "temp_rational". While "get" returns positive results or "+", "add" alternately sums rational values

to sum. A negative result from "get" stops the magic, and yields either the correct result, or "error". As before, a period ends the file.

*** code segment ***

```
1: add :: {sum = 0
2:         addterm:
3:         {get    $ {  sum += temp_rational;
4:                      get == '+' $ add_term\
5:                      } addterm ..
6:          put "error" ..
7:          }
8:         put sum
9:         }.
```

*** explanation ***

1: The function name is "add", and the first statement in the defining block initializes the temporary value "sum" to zero.
2: The next block, starting on line 3: is labeled "addterm".
3: The new block is opened, and the function "get" is called to give a new value to "temp_rational". If a rational is available the unlabeled block is opened, and "temp_rational" is added to "sum". Otherwise control passes to line 6:.
4: If get returns a '+' control passes to the label "addterm", or line 2:. Otherwise control continues to line 5:.
5: Control passes to the line following the block labeled "addterm", line 8:. The current unlabeled block is terminated.
6: The quoted literal "error" is sent to the output device, and control passes to the bottom of the block labeled "add_term" (7:).
7: The current block is closed. (add_term)
8: The result "sum" is sent to the output device.
9: The current block is closed (add), and the actor definition file terminated.

Before "add" can be used its declaration, and the declarations of the objects it requires, must be accomplished. Note that "+=" was defined once, but appears twice!

*** code segment ***

```
 1: storage rational_number_declarations
 2: share rational_number_definition,
 3:       rational_actor_definitions,
 4:       standard_definitions
 5: rational: sum, temp_rational,
 6:           + rational, +: rational, reduce rational
 7: integer:  i,
 8:           +: integer
 9: :         add            -- no arguments required
10: .                       -- none returned
```

*** explanation ***

1: The "storage" file is named "rational_number_declarations".
2: The scope is opened to include "rational_number_definition",
3: "rational_actor_definitions", and
4: "standard_definitions".
5: The passive objects "sum" and "temp_rational" are declared to be of type rational.
6: The actors, '+', "+:", and "reduce" are declared to have rational arguments and results.
7: The object 'i' is declared to be an integer, and
8: the operator "+:" is declared to have integer arguments and results.
9: The function "add" is declared to require no arguments and to produce no results.
10: The period terminates the storage declaration file.

All that remains is to run the program. Typing at the prompt yields the following results.

```
>add 1/12 + 1/4 + 1/6
1/2
>1/12 + 1/4 + 1/6
1/2
>
```

The latter does not use the function "add", but the '+' operator directly. If the input line was typed without the rational definitions, the result would be different (i.e. 0).

This concludes the introductory example, and should have convinced even the casual reader that the proposed language D is, at the least, <u>different</u>.

V. LANGUAGE DESCRIPTION

An enumeration of the notation selected to describe the language will precede the language description. The language will be described from the bottom up, starting with the symbols constituting the language and concluding with the structured extensibility embodied within it.

Notation

Notation used to convey the language syntax consists of an alphabet of nonterminals, terminals, and punctuation.

Nonterminals are symbols which are defined by productions included in the syntax, and are strings of printed characters bracketed by '<' and '>'. Terminals are strings of printed characters which stand for themselves, and appear in D programs. Productions show how nonterminals can be reduced into simpler nonterminals and terminals, and are of the form:

```
<file> ::=
        <state_file>    | <actor_file>      |
        <storage_file>  | <bundle_file>
```

The punctuation '::=' indicates that the production reduces the nonterminal <file>. Vertical separators, '|', separate alternatives. Wherever "<file>" appears, any of the four alternatives listed above can be substituted. The primary units of punctuation used in the notation describing the language are "<,>,::=,|".

To reduce the amount of text needed, additional punctuation is introduced. Text between dollar signs, "$<text>$", is optional. Text between pound signs, "#<text>#", must be repeated one or more times. Parentheses are included to reduce the need for intermediate

nonterminals. For example:

```
<literal> ::=     (<digit> | _ ) $#<char>#$
```

could also be written:

```
<literal>  ::=     <digitor_>$#<char>#$
<digitor_> ::=     <digit> | _
```

but would require the additional nonterminal <digitor_>.

In several cases, the parentheses bracket a single nonterminal, and in these cases the parentheses are terminals; not punctuation. Note that the construction "$#<text>#$" denotes zero, one, or more repetitions of "<text>". The secondary units of punctuation used in the notation describing the language are: "$,#,(,)".

If lexical units are equivalent syntactically, but are semantically different the construction is as follows:

```
<array_name>          |
<file_name>           |
<funct_name>          |
<label_name>          |
<pointer_name>        |
<scalar_name>         |
<structure_name>       ::= <name>
```

This conveys that the seven left-side nonterminals are syntactically equivalent to the right-side nonterminal.

This concludes the introduction to the notation used in this document to describe the D syntax.

Symbols

There are five types of tokens in D: symbols, names, quoted literals, integers, and comments. Symbols will be described first.

The language alphabet consists of four disjoint sets of symbols, which can be considered tokens which uniquely affect a user output device (such as a Video Display Terminal (VDT)). They are character, operator, punctuation and format symbols.

Characters. Characters consist of the upper and lower case alphabet, (a..z, A..Z), the digits 0..9 and the underscore character '_'. Characters are used to form names and literals. They can appear in comments and quoted literals.

```
<char> ::=
        <digit>     | <alpha>       | _

<digit> ::=
        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<alpha> ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M |
    N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
    a | b | c | d | e | f | g | h | i | g | k | l | m |
    n | o | p | q | r | s | t | u | v | w | x | y | z
```

Operator Symbols. Each operator name consists solely of operator symbols. The operator symbol set defines the symbols that can be used to form an operator name, and define the precedence of the operator. An operator's precedence is determined by the right most operator symbol of its name. The following production defines the precedence associated with each operator symbol. Starting from the left, the first two operators (!,^) have the highest precedence. Each group of two symbols to the right has a lower precedence than the two to their left. The assignment operator (=) has the least precedence.

```
<oper> ::=

        ! | ^ | * | / | ~ | % |
        + | - | & | | | < | > | =
```

Operator symbols can appear in comments and quoted literals.

Punctuation. The syntax of the language is vested in the punctuation, which consists of 20 symbols; each have a unique meaning which may be context dependant. Punctuation can appear in comments and quoted literals.

```
<punct> ::=
        { | } | [ | ] | ( | ) | , | : | :: | ~~
        ` | ' | " | @ | ? | # | $ | ; | \ | ..
```

The following table describes the the meaning that each of the punctuation symbols can have in each of the three punctuated files; 'S' corresponds to the State file, 'A' corresponds to the Actor file, and 'D' corresponds to the storage Declaration file. A 'e' in a file, (S or D), position indicates that the symbol is introduced only through expressions. Similarly, a 'd' in a file, (S or A), position indicates that the symbol is introduced only through declarations.

Table I: Description of Punctuation Symbols

symbol	S A D files	meaning
{}	S	delimits structure definitions
	A	delimits blocks
[]	S A D	denotes an array by following a <name>
	d d D	with an index range optionally specified by a scalar value within the square brackets
	e A e	with an index optionally specified by a scalar value within square brackets
()	e A e	controls expression precedence
)	S	suppress zeros to right
(	S	suppress zeros to left
,	S A D	line continuation character
:	d d D	<class> delimiter -- declaration
::	S A	<class> delimiter -- definition
--	S A D	comment initiator
`	e A e	single symbol quoted literal delimiter
"	e A e	string quoted literal delimiter
'	e A e	possession indicator
#	d d D	creates pointer to declared class
	e A e	obtains object referenced by pointer
@	e A e	address abstraction on following <name>
$	d d D	class abstraction on following <name>
	A	delimits guard expressions
?	A	selects block behavior
<blank>	A	goto next line of block
;	A	goto next open line of block
\	A	goto beginning of block & reexecute
..	A	goto end of block

KEY: S - contained in state file
A - contained in actor file
D - contained in storage (Declaration) file
e - only appears in expressions
d - only appears in declarations

Keywords are also a a form of punctuation, and keyword names are reserved. The language keywords perform three specific functions: they determine file type (actor, bundle, state, structure); they denote interfile visibility (copy, rename, share); and they are used to define actors. In the latter role, they control intertask communication (accept, in, out, return) and dynamic memory allocation (new, null).

<keywords> ::=

accept	new	return	
actor	null	share	
bundle	out	state	
copy	rename	storage	
in			

Format. Formatting symbols organize other symbols upon the page and delimit files. Horizontal white space (consisting of blank and horizontal tab tokens) is insignificant except to delimit names. Vertical white space (consisting of line feed, vertical tab and form feed tokens) serves to end lines, unless the last nonformatting and noncomment symbol is a comma, ','. Commas serve to continue the current logical line on to the next physical line. All files are terminated with a period '.'.

Formatting symbols can appear in comments and quoted literals, except for line terminators (vertical white space). The printable symbol set consists of the above mentioned symbol set, including the white space formatting symbols.

```
<format> ::=    <wspace>        |
                <fterm>         |
                <lterm>

  <wspace> ::=  <HT>            |
                <SP>

  <lterm> ::=   <LF>            |
                <VT>            |
                <FF>

  <fterm> ::=   .

<psymbol> ::=   <punct>         |
                <oper>          |
                <char>          |
                <wspace>
```

Naming

A name can be loosely considered as a string of symbols with which the user identifies labels, functions, operators, scalars, structures, pointers, arrays, files and classes. Syntactically, there are three classifications of identifiers.

Names. In the context of the language, the term name has a specific meaning. It is an <alpha> followed by a string of characters. Any number of characters can be used within limits. Two limits immediately come to mind: comprehensible limits, and line length limits. If commas are used, note that they continue lines but delimit names. Only operator names may not be derived from this classification.

```
<name> ::=      <alpha>$#<char>#$
```

Literals. The second kind of name is what will be called a literal. It is identical to a name except the first character must be a digit or an underscore (_). Such an object may only be assigned a value by initialization during declaration. Its value from then on is constant. Only operator names may not be derived from this

classification.

<literal> ::= (<digit> | _)$#<char>#$

Operator Names. The third kind of name is an operator name which is constituted by 1,2 or 3 operator symbols. It denotes, with the argument and result class, a particular operator. Like functions, operators are user definable. Unlike functions, which may take one argument and may return a result, operators require one or two arguments and must return one result.

<op_name> ::= $<oper>$$<oper>$<oper>

Quoted Literals

For situations where the value of an expression is a string of symbols, as opposed to a defined internal value, two kinds of quoted literals are provided. Quoted literals are represented as a symbol, or symbols, delimited by a reverse appostrophe '`', or quotation marks '"', respectively.

Formatting symbols should be provided with a printable representation for inclusion within quoted literals.

<qliteral> ::= `<symbol>` | "#<symbol>#"

Integers

The only predefined type in the language is integer, which may be denoted int. Scalar types are defined by creating finite mappings onto a contiguous set of integers. A string of digits and underscores constitititute a literal which denotes the expected value. Underscores are not significant although it is suggested that they be used to separate chiliads, or groups of three decimal digits.

Any expression which yields, as its result, a scalar is syntactically an integer.

```
<int>   ::=     (#<digit>       | _ #)          |
                <scalar_expression>
```

Comments

Comments may be inserted into the language following the token ("--"). Text on the remainder of the line will be rendered syntactically and semantically insignificant. The comma acts to continue comments as it continues lines. If both a line and an comment are continued, the next physical line contains the continuation of the comment. The only symbols which cannot be included in comments are the terminator symbols for lines.

```
<comment> ::=   --$#<psymbol>#$ <lterm>
```

Expressions

Functions, objects, and literals are all sources of values. Expressions connote values as static objects denote them. Function names always precede their single argument, and bind more tightly than operators. Operators permit multiple values to be reduced to a single value. Binary operators require two value arguments, one on either side. Unary operators may either precede or follow their argument. Operator precedence is determined by the right most operator symbol of the operator name. Parentheses can be used to change the precedence of expression evaluations.

The assignment operator takes two arguments, a logical name and an expression. The assignment operator has the lowest evaluation precedence and returns the the value of its right hand side argument.

Multiple logical names may be assigned values in one expression.

The value produced by the expression, right side, is converted to the class of the object referenced by the logical name, left side, with a user supplied conversion routine and then assignment is accomplished. Conversions may not always be required. The assignment is defined in this way for all user defined classes.

Logical names may denote the values of the objects they reference and may consequently appear in expressions. A '@' preceding a logical name returns, as a value, a logical name. This value is equivalent to that held by a pointer to the object referenced by the logical name.

The order of subexpression evaluation is undefined. For example the evaluation of the expression

$$f(A) + (f(B) + f(C))$$

may start with the evaluation of f(A), f(B), f(C) or simultaneous evaluation of all three functions. If the evaluation order of subexpressions is important explicit temporary variables should be used to force the order of expression evaluation.

Portions of an expression may be evaluated by different tasks upon different processors. Expressions are the basic unit of parallel execution within the language. Evaluation of an expression will cause the owning process to hang until all the values required for its evaluation are available and all subexpression evaluations have been accomplished. The block structure will be introduced below, permitting neutralization of expressions which tend to hang.

The keyword accept has been integrated into the language to refer to a function that can intercept actor calls. It captures the actor's

arguments, if any, and the caller identification. Arguments to an accept function consists of a function or operator name.

The value returned by an accept expression to its containing expression is either the function argument it is passed or the value denoted by the keyword null signifying that a request for services has been made but no argument was passed. The caller identification is passed to the next return statement, discussed below, in the thread of control.

The keyword null is also used to denote the value of a pointer which does not reference an object.

```
<expr> ::=      <literal>                 |
                <qliteral>                |
                @<lname>                  |
                <lname>                   |
                <expr><op_name>           |
                $<expr>$<op_name><expr>   |
                (<expr>)                  |
                <funct_name>$<expr>$      |
                accept <funct_name>       |
                null
```

The concept of a logical name permits values to be accessed using more than one method. Usually the object is directly named. If the object is part of a structure, the structure and the component separated by a (') constitute the logical name. Pointers provide indirect access, and arrays provide indexed access. A pointer name points to, or has as a value of, the logical name of an object. A '#' immediately preceding a pointer name provides the value of the object. An array name returns an array; components may be accessed with indices. Multiple levels of pointers and multidimensional arrays are possible, but each level must be explicitly declared one at a time. The class abstraction operator permits an "array of arrays", a "pointer to a pointer", or a "pointer to a function" to be declared.

The number of meaningful '#' preceding a logical name is dependent upon the referencing path of the logical name. Too many result in the null value. File names may also be logical names.

```
<lname> ::=     <object_name>                    |
                $#$<lname>                       |
                <lname>$[($<int>$)]$             |
                <file_name>                      |
                <lname>'<lname>
```

Statements

The statement is the basic unit of sequential execution in the language. Once a statement has begun execution it must completely finish before control is passed to another statement. There are four kinds of statements; three of which can be interpreted as expressions and one which is a compound statement or a block.

Nominally each statement consists of an expression which is evaluated. In addition to expression evaluation, the language supports interprocess communication, local access to external modules, conditional execution, and dynamic memory management.

```
<statement> ::=

        <expr>                                   |
        return $<expr>$                          |
        <block>                                  |
        new <pointer_name>$=<expr>$
```

If an operator or function returns a value, a formal variable must be declared by the function definition to hold the value to be returned when the operator or function is terminated. A return statement may be considered an expression consisting of the language defined return function. In any case the expression, if any, in the return statement is assigned to the formal result object, and the enclosing actor is terminated.

Following an expression which includes an accept function, the return statement sends its argument back to the calling function. The accepting actor is not terminated because its activity does not stem from the calling process, but from some other source.

Another statement that can be considered an expression consists of the keyword new followed by a known logical name which has been declared a pointer to a class C. This language defined function dynamically creates another instance of an object of type C, and assigns the value of its logical name to the pointer. This new object may be initialized.

Objects which are no longer referenced by any pointer cease to exist. If immediately after creating an object with new, the pointer used is assigned null, the object would eliminated and its resources reclaimed. A function 'free' could be written by tracking down all references to a object and setting them to some other object, or null.

Objects dynamically created and shared with other processes may be referenced by object pointers outside of the control of the creating process. Until all the objects referencing an object are terminated, the object cannot be itself terminated.

Statements are the element of sequential execution in the language. A block is a compound statement which provides some relief from this monotony. It contains mechanisms to locally introduce objects externally declared, and to implement conditional and iterative execution of statements.

Blocks

In addition to serving syntactically as a statement, a block is the basic unit which defines each actor. Statements can only appear within blocks.

A block consists of an optional header followed by linkage and a a list of lines. The block is the fundamental unit of scope control. The block is the smallest syntactic unit within the language to permit local names to be introduced via linkage. A block, with the exception of an optional header, is delimited by curly brackets ({,}).

```
<block> ::=     $<header>$
                {<linkage>
                #<line>#
                }
```

Each line consists of a statement optionally prefaced by a guard, and optionally succeeded by a continuation. The guard supports conditional execution, and the continuation supports a limited jump capability.

```
<line> ::=      $<guard>$<statement>$<contin>$<lterm>

  <guard> ::=   <expr>$

  <contin> ::=  $<label>$       |
                $<label>$\      |
                $<label>$..     |
                $<label>$;
```

After the statement has been executed, the continuation determines which statement will execute next in conjunction with the guards. The enclosing block referred to by the continuation is identified by the label. No continuation character indicates that the next statement will be executed regardless of whether the guard is open, closed, or not yet evaluated. A backslash (\) indicates that the block will be reentered. Two double dots (..) indicates that the block will be terminated. A semicolon (;) indicates that the next open statement of the block will be executed. From the top of the block, control first must pass to an statement with an open guard. Statements without guards are considered open.

Each time a block is entered the vector of guards is evaluated and the results stored in a vector of guard values. These values will be maintained until the block is reentered or terminated.

The header consists of a label and an optional selector. The label identifies the block for the continuation as just described.

```
<header> ::=

        <label>:  $<selector><lterm>$

  <selector> ::=

        ?               |
        <expr>          |
        ?<expr>
```

The selector consists of an optional expression and and optional token (?). The expression, if present, is evaluated each time the block is entered. The resulting value is the value that open guard expression must have. If the expression is absent, then guards with values greater than zero are open.

The token (?) is used to modify the sequencing behavior of the block. Nominally, no (?), the guard vector is sent off for evaluation. Each guard is evaluated simultaneously with the other guards. Each guard value is assessed, in the order the guards appear in the block. Control passes through the first open guard to the guarded statement. If the block is reentered before all the guards have been evaluated, more guards clog the evaluation stack. However, all guard evaluations must be complete before the block can be terminated.

The alternate form of block control, indicated by the presence of a (?) following the label, sends the vector of guard expressions out for evaluation simultaneously as before. In this case control passes through the first guard which is returned open --- regardless of

position in the block. When the block is reentered or terminated, all pending guard evaluations are flushed.

Scalars and Structure Definitions

In order to apply algorithms in the form of functions and actors to objects, the precise characteristics of objects must be defined. Unlike most widely used languages, this language leaves such definitions for the users (or programmers).

There are four kinds of definitions which occur in two kinds of files. A state definition file is used to define the static objects called scalars and structures. An actor definition file is used to define dynamic objects called operators and functions. This subsection will describe the definition of static objects, and the next subsection will describe the definition of active objects.

Each definition of a static object assigns to a scalar or structure class: 1) a name; 2) representation internal to the language; and 3) a convenient user representation. Scalar and structure definitions constitute state definition files.

Each state definition file starts with the keyword state which is immediately followed by the file name. Linkage follows. Linkage is followed by the list of definitions. Scalars are defined to be simple objects which can assume integer values. Structures are declared to be composites of scalars and other structures.

<state_file> ::=

state <state_file_name><lterm>

<linkage>

(#
<scalar_definition> |
structure_definition>
)#
.

The internal representation of scalars is based upon integers. The value of each object may be constrained between bounds, and must be explicitly integer valued. Derived classes are function and operator compatible with actors compatible with their parent class.

If a single integer is specified, the range implied lies between the value of the specified integer and zero. Nominally scalar values range from 0 to a single upper bound. If two bounds are given, the scalar value ranges from the first, lower bound, to the second, upper bound, inclusive.

<scalar_definition> ::=

<scalar_name> :: <int>$..<int>$ <lterm>

Scalars are implemented in precisely as many bits, in a binary representation, as is required to contain the range desired. Other nonbinary implementations are possible. Implementations may suggest ranges for efficiency. The user sees the values of scalar objects as integer literals. Negative values are preceded by a '-'. Classes given a range permitting negative values hold a space for the sign.

Structures enable compositions of all previously defined classes to be bound together. Recursion is not permitted, but pointers within a structure may point to the objects of the structure class being defined.

<structure_definition> ::=

<structure_name> :: {#<comp><lterm>#

}<lterm>

Each structure is defined as a list of composites. A composite is either an object declaration or a a string of symbols. Parenthesis may be placed on either side, or both sides, of the symbol string.

The internal representation of directly referenced objects on the structure list is their scalar representations as described, from top (left) to bottom (right), tightly packed. Pointers and arrays form boundaries over which this need not be true.

The user I/O representation consists of the component scalar values presented from left to right, modified by interspersed symbol strings in the structure definition list. A '(' or ')' suppresses zeros beside the symbol string, on the side it appears.

<comp> ::=

<object_decl>
$($ (# <psymbol> | . #) $)$
$($)
(

Function and Operator Definitions

Functions and operators are objects which act upon other objects. They are defined in a file initiated by the keyword actor, followed by the actor file name. The actors are defined in terms of previously defined objects, and accessed through linkage.

In addition to a name and linkage, each file consists of a list of definitions. Each definition consists of a name and a block, and may include a result and an argument declaration. The manipulations it performs are specified by its block.

```
<actor_file> ::=

        actor <actor_file_name><lterm>

        <linkage>

        (#

        (  <op_decl>     |
           <funct_decl>     )

        <block><lterm>

        #).
```

The class and number of arguments and results of functions and operators must be declared in their definitions. The class can be declared generic. The declarator determines whether an array, pointer, or object value is required or returned. The structure of the argument is provided as an object declaration following the keyword in. The structure of the returned value is provided as an object declaration following the keyword out.

An operator must have one or two arguments. An operator must always return a result. The form of the in declaration contains two declarators. The first corresponds to the argument preceding the operator. The second, following a comma, corresponds to the argument following the operator. The comma must always precede the following argument.

If the input argument is initialized, the operator may optionally omit the argument, the initialization serving as a default. Operators must have one uninitialized declarator. If the output is initialized, and the operator fails catastrophically during execution, the initialized result is returned.

```
<op_decl> ::=

   <op_name> :: in  <class>:
                    (
                    <declaration>, <declaration> |
                    <declaration>                |
                    ,<declaration>
                    )<lterm>

                out <class>: <declaration><lterm>
```

Functions are similarly defined, and may have only one argument. The argument and the result are optional. Both the argument and the result may be initialized.

```
<funct_decl> ::=

    <funct_name> :: $out <class>:
                         <declaration><lterm>$

                    $in  <class>:
                         <declaration><lterm>$

                         <lterm>
```

Storage Declarations

A storage file has a name preceded by the keyword storage and consists of linkage followed by a list of object declarations.

```
<storage_file> ::=

        storage <storage_file_name><lterm>

        <linkage>

        #<object_decl><lterm>#

        .
```

In this file, objects are created from defined templates identified as classes which are defined in state definition files.

<object_decl> ::=

<class>: ($#<declaration>,$#) <declaration>

<class> ::=

<file_class> | <object_class>

Classes may refer to either objects or files. Files are a special form of object containing modules of the language. File classes consist of state files, actor files, storage files and bundle files.

```
<file_class> ::=    state   | actor         |
                    bundle  | storage
```

Object classes are defined by scalar definitions, structure definitions, operator definitions, and function definitions. Scalars and structures can be specified directly by definition names. All classes may be obtained indirectly by abstracting the object class from an object referenced by a logical name with the class abstraction operator ($).

```
<object_class> ::=      <scalar_name>   |
                        <struct_name>   |
                        <op_name>       |
                        <funct_name>    |
                        $<lname>
```

Declarations instantiate objects from visible definitions. Upon instantiation of an object according to a named definition, a referencing method must be established for each object, and the object may be initialized. If the created object is a constant, as indicated by its name, it must be initialized. An initialization is an expression of already known objects which returns a value of the class of the object declared. This value is assigned to the declared object. Structures cannot be initialized component by component.

<declaration> ::= <declarator> $= <expr>$

The declarator defines how the object will initially be referenced. If the declarator is a name, a direct reference is provided.

If the name is preceded by a '#' then the name itself provides an indirect reference to the object and is called a pointer. The declaration creates a variable that holds, as a value, the logical name of a object of a particular class. Pointers can be initialized to logical names of objects with a class compatible with the declaration. The pointer name references a logical name as a value. When the pointer name is preceded by the '#', the logical name held as a value by the pointer is used to reference an object.

If the name is followed by square brackets, relative addressing is indicated. The name alone provides the logical name of an entire array. The name with an index 'n' provides the logical name of the n'th object in the array. If a scalar appears between the square brackets in a declaration, it serves as a bound upon the range of possible indices. If the brackets are empty, no constraints are imposed. In this context, all scalars are converted to equivalent integers internally.

The pointer symbol '#' binds most tightly, followed by the square brackets indicating an array. Precedence in a declaration may not be changed with parentheses. The logical name "#array_of_pointers[n]" references an array of pointers, <u>not</u> a pointer to an array.

Each active object acts upon objects of one class, and returns values which may be compatible with another. Active object declarations may be contained within a list of passive object declarations returning objects of the same class. The class of the value returned is analogous to that returned by passive objects. The class of the argument, if any, must follow the operator or function name.

<declarator> ::=

<object_name> |
#<pointer_name> |
<array_name>[$<int>$] |
<op_name> <class> |
<funct_name> <class>

Array names and pointer names can both be recursively defined during declaration, but not within a declarator. A series of declarations using the class abstraction operator ($) must be made, first declaring a pointer 'P', declaring 'P1' to be a pointer to the class of 'P' etcetera.

File Organization

The building block through which most algorithms should be implemented is the module, or file. The language supports four file types: two to define objects, one to instantiate objects, and one to organize the special objects called files.

Each file begins with a descriptive keyword unique to it, which describes its function. Following the keyword is the name of the file object. On following lines, linkage is included which defines the context of the body of the file.

The language permits and facilitates modulation of algorithms, definitions, objects, and communication channels. Linkage permits familiar concepts to be placed into new contexts to serve in new capacities. In the context defined, the body of the file accomplishes the definition, declaration or juxtaposition dependent upon file type. Each file ends with a file terminator which is the same for all files.

Linkage

Execution requires that several modules work together to provide meaning to a program. Linkage provides a means for one module to access another by either copying a module for exclusive use or shareing access with other modules. It also permits local logical names and object class names to be bound to different names defined within the accessed modules.

First, all the files which can be shared are listed following the key word share. Second, all the files which must be copied are listed following the key word copy. Third, the key word rename is used to introduce bindings of new names to an external object names and class names.

```
<linkage> ::=

    $share (#$<file_name>, #$)<file_name><lterm>$

    $copy  (#$<file_name>, #$)<file_name><lterm>$

    $rename #(      <lname> <name><lterm>        |
                    <object_class> <name><lterm>
            )#$
```

Linkage can be used to hide currently visible names. Renaming takes place before the linking file knows about the linked object name: the old name has no effect on the linking file's name space. Objects can only be renamed in the original linking operation.

Storage files, which contain the values of variables, are usually copied because these files are modified during the course of program execution; unless interprocess communication is desired. Definition files are usually shared, because they are not usually modified. Since the copy operation performs object instantiation, each file may have its own private cache of objects.

Bundles, or Encapsulated Linkage.

A module, or file, in this language has no intrinsic meaning. A collection of modules is required to define object characteristics, and to declare objects. Several modules, together, constitute an instantiation of an automated algorithm. A bundle of modules forms an element which can, with other elements, constitute an application package.

```
<bundle_file> ::=

        bundle <bundle_file_name><lterm>

        <linkage>

        #<file_name><lterm>#

        .

  <file_name> ::=

        <state_file_name>       |
        <actor_file_name>       |
        <storage_file_name>     |
        <bundle_file_name>
```

Bundle files may contain references to other bundles, but not to themselves.

A library of modules which do not constitute a complete algorithm may be bound into a bundle for convenience.

VI. ALGORITHM EXAMPLE: A Vector Dot Product

In general, computers are useful because they eliminate errors by performing consistency checks upon assumptions. Put another way, a major use for computers is to perform simulations based upon a set of assumptions. The computed results determine if the assumptions are consistent with the expected results. A reasonable objective is to communicate with computers in a language which supports as high a level of abstraction as possible. First, because the domain of trivial errors is minimized; and second because communication efficiency is increased.

The purpose of this section is to show how the proposed language can be easily extended to efficiently support ideas specific to particular users; in this case the dot product operation applied to vectors. This section consists of an example of how the language may be formed to fit particular users' needs, how parallelism is supported, and how algorithms can be generically specified. This example is a microcosm of the computer capabilities the proposed language has been developed to support.

After reviewing the mechanics of the dot product operation, the file hierarchy required to support a generic encoding of the dot product algorithm, in D, will be described. In addition to code implementing the dot product algorithm, code to define a vector, vector components, and operations upon vector components will be listed. The effect that special purpose hardware might have on speeding the algorithm will be discussed, followed by a summary of the key points of this section.

The Vector Dot Product

For the purposes of this thesis, a vector can be considered as a one dimensional array of 'n' components, where 'n' is a positive integer. Neither the class of the component nor the specific value of n need be explicitly specified.

The vector dot product, for the purposes of this example named '*', requires two vector arguments. Each vector must consist of 'n' components of the same class. The operation forms a value with the class of the components.

If A = (a1, a2, ... an) and B = (b1, b2, ... bn) then A*B = (a1*b1 + a2*b2 + .. + an*bn), where the component operation '*' would nominally be a scalar multiplication for integer ai and bi.

The File Hierarchy

To completely define a vector dot product algorithm for a specific vector, additional information beyond that provided by the generic algorithm specification, just given, is required. Characteristics of the components need to be defined, as must a set of compatible operators. The representation of a vector must be specified, and its length must be set. This done, a dot product can actually be computed.

Figure 1 shows how the vector dot product is constructed from the basic standard definitions. The file class is written above the box. Some boxes represent multiple file classes. The number inside the box, in parentheses, is provided to connect the figure with the following code listings. Asterisks, '*', indicates code for the block is subsequently listed.

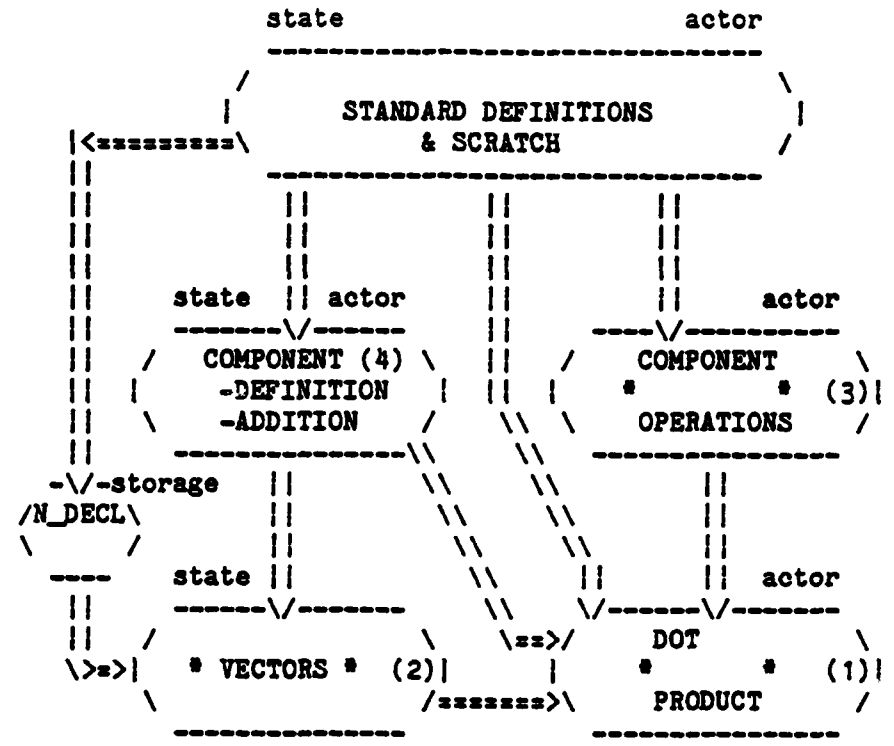


Figure 1: File Hierarchy

The Vector Dot Product Algorithm (1)

The following generic dot product algorithm first establishes the other files required to give the definition meaning. Some simple error checking is defined, on errors passing control to functions which are implicitly visible. The result "dp" is cleared and the index 'n' is initialized.

Then the dot product is computed. The block structure spools the component expressions onto the execution queue exactly 'n' (the number of components) times. The variable "dp" collects the values of all the queued expressions. The block terminates when all the values have have been collected. Since the expression queue is a parallel structure, parallel execution will occur if there is sufficient hardware.

To conclude, the result is returned to the task that called the dot product operator. The listing follows:

*** code segment ***

```
 1:  actor dot_product                                  ~~ (1)
 2:  share vector_definition,
           component_definition,
           component_addition,
           error_definition
 3:  copy component_operations, scratch
 4:  #    ::  in  vector: v1,v2
 5:           out $v1'v[]: dp

 6:  block1:
 7:  {    v1'n != v2'n      $ dimension_incompatible..
 8:       v1'n == 0         $ dimension_zero..
 9:       n = v1'n
10:       dp = 0
11:       block2:
12:       { n > 1                              $ n = n-1
13:         dp = v1'v[n] * v2'v[n] + dp $ ..
14:         }
15:       return dp
16:       }.
```

*** explanation ***

1: This is an "actor" definition file named "dot_product".
2: Access is required to a file named "vector_definition", "component_definition", "component_addition", and "error_definition".
3: Exclusive access is required to files named "component_operations", to give meaning to (!=, ==, >, *, +, -) and provide for integer to vector component conversions, and "scratch" to declare and hold the value of the object denoted by 'n'.
4: The generic actor (dot product operator) is assigned an operator name '*', and two formal arguments "v1" and "v2" are defined to be of the class "vector".
5: The result, named "dp", is declared to be of the same class as the vector components.
6: The defining block is labeled "block1".
7: Block1 is opened, and the dimensions of each vector are checked for compatibility. If they contain a different number of components, then control passes to the function "dimension_incompatible" defined in the file "error_definition" and to the end of the block (16:). Otherwise, control passes to the next line, (8:).
8: The arguments are checked to insure they have a nonzero dimension. If their dimension is zero, control is passed to the function "dimension_zero", defined in the file "error_definition", before terminating the block as above.

9: The dimension of the arguments are assigned to the scratch variable 'n'.
10: The result is initialized to zero.
11: The following block is labeled "block2".
12: The block is opened, and all the guards are evaluated with the current value of 'n'. As soon as 'n' is determined to be greater than 1, 'n' is decremented and the cycle repeats. When "n<=1", the block terminates.
13: Each time the block is opened, a new value is added to "dp" which is dependent upon 'n'. The block may be reentered before all the guards have been evaluated from the last entry, demonstrating a decoupling between the control and expression evaluation parts of the language. This may be exploited via parallelism to improve execution speed.
14: Block 2 is closed after all guard expressions have been evaluated.
15: The value contained in the output variable, "dp", is returned to the calling actor.
16: Block1 is closed, and the actor definition file terminated.

Definition of the Class "Vector" (2)

A vector is implemented as a structure with two components. The first component is the dimension of the vector; an object which stores the number of objects in the array. The second component is the array of components.

The value of 'n' must be known, and accessible to this definition, before this vector definition can be invoked. It is provided in the file "n_declaration", which is assumed to exist. The file required to define a vector follows.

*** code segment ***

```
1:  state vector_definition                    ~~ (2)
2:  share n_declaration
3:  share standard_definitions
4:  vector :: { int: n
5:              component: v[n]
6:              }.
```

*** explanation ***

1: This file will define static classes, and is named "vector_definition"
2: The file "n_declaration" is required to determine the number of components.

3: The file "standard_definitions" is required to define the class "int", although it could be deferred as is "component".
4: The structured object to be defined is named "vector", and the first component of the structure is named 'n' and is of class "int".
5: The second component is an 'n' dimensional array of "components" named 'v'.
6: The structure definition and state file are terminated.

Components and Compatible Operators (3 & 4)

Components may be defined by considering a passive object definition and basic set of operators, and then the more general set of component operators required in the dot product definition.

First consider the definition of the components and a basic set of operators (+, -, =) (4). The '=' operator is predefined for all classes, but the meaning of the binary '+' and unary and binary '-' needs to be explicitly specified. In addition it is convenient to specify component to integer and integer to component conversion procedures in the form of functions with the class names and arguments of the class to be converted. This done, literals composed of digits take on their usual meanings.

Since the object is a generic algorithm the component definitions, just described, will be assumed to exist. The component class rational has been defined, see section IV, and could be used here.

Second, consider expanding the kernel set of operators in terms of themselves (3). In this case the operators (==, >, *) are required since they were used in the code above. They are defined below.

*** code segment ***

```
1:  actor component_definitions                  -- (3)
2:  copy component_definition
3:  copy component_addition  -- (+, -, int <=> component conv. )
4:  copy scratch             -- (sign, n)
```

```
 5:  == :: in  component: operand1, operand2
 6:        out component: result
 7:        { result = operand1 - operand2;
 8:          result  $
 9:          -result $ result = 0 ..
10:          result = 1 ..
11:          }

12:  >  :: in  component: operand1, operand2
13:        out component: result
14:        { result = operand1 - operand2;
15:          result $ result = 1 ..
16:          result = 0 ..
17:          }

18:  *  :: in  component: operand1, operand2
19:        out component: product
20:        { product = 0;
21:          sign = 1;
22:          -operand1 $ { sign = -sign
23:                        operand1 = -operand1
24:                        };
25:          -operand2 $ { sign = -sign
26:                        operand2 = -operand2
27:                        };
28:          operand1 == 0 $ product = 0 ..
29:          operand2 == 0 $ product = 0 ..
30:          { operand1 = operand1 - 1 $ \          -- parallel
31:            product = product + operand2 $ ..
32:            };
33:          -sign $ product = -product;
34:          }
```

*** explanation ***

1: The file contains actor definitions and is named "component_definitions".
2: The files defining the structure of a component, "component_definition",
3: and defining the basic set of operators (+, =, int <=> comp.), "component_addition", are required to be available.
4: A "scratch" file must also be visible in which the variables "sign" and 'n' are declared.

5: The "==" operator is defined to take two operands of class "component", with the formal names "operand1" and "operand2".
6: The result will also be of class "component" and has the formal name "result".
7: The formal object "result" is assigned the difference between the two operands, and control is passed to the next statement with an open guard.
8: If "result" is positive, control passes to the next statement despite the guard value. If "result" is negative, control passes to (9:) and the value of the guard is considered.

```
 9: If "result" is negative, or considering the previous statement
    nonzero, the value of 0 is assigned to "result" and the block
    is terminated.
10: Otherwise, "result" is assigned the value 1,
11: and the block is terminated.

12: The '>' operator is defined to require two operands of the
    class "component" which are given the formal names "operand1"
    and "operand2".
13: The result is named "result" and is of the class "component"
    as well.
14: The defining block opens by assigning to result the difference
    between "operand1" and "operand2". Control passes to the next
    open statement: (15:) if "result" is positive.
15: If "result" is positive, it is assigned the value 1 and the
    block ends causing the operator to return the value of
    "result".
16: Otherwise, result is assigned the value 0,
17: and the operator '>' returns the value of "result".

18: The '*' operator is defined to require two operands of the
    class "component" which are given the formal names "operand1"
    and "operand2".
19: The result is of class "component" with the formal name
    "product".
20: "product" is initialized to the value 0, and control passes to
    the next statement without a closed guard.
21: "sign" is initialized to the value 1, and control passes to
    the next statement without a closed guard.
22: If "operand1" is negative, control is passed to the unlabeled
    block, "sign" changes sign, and control passes to the next
    statement (23:). Otherwise control passes to the next
    statement without a closed guard.
23: "operand1" changes sign.
24: The current block ends, and control passes to the next
    statement without a closed guard.
25: Similar to (22:) with "operand2" replacing "operand1".
26:   "     "  (23:)  "      "          "         "
27:   "     "  (24:)  "      "          "         "
28: If "operand1" is equal to 0, "product" is set equal to 0, the
    block is terminated, and "product" is returned to the calling
    actor.
29: Similarly, "operand2" equal to 0 causes the same effect.
30: A new block is opened, and the guard vector is evaluated.
    "operand1" is decremented, and "operand2" is accumulated into
    "product". If "operand1" is greater than zero after it has
    been decremented, then control passes back to (30:) for
    another guard vector evaluation. When "operand1" finally is
    nonpositive, control is permitted to pass to the next
    statement.
31: Each time the block is entered, the guard expression is placed
    into the evaluation queue. Once the iteration is done,
    control passes either way through this statement to
```

32: close the current block. Control passes to the next statement without a closed guard.
33: If "sign" has a negative value, then the sign of "product" is changed.
34: The defining block is closed, the file terminated, and "product" is returned to the calling actor.

For completeness, the division operator is also defined in terms of the kernal operators without elaboration. The working part of this algorithm must be executed sequentially in contrast to the multiplication algorithm in which portions could be evaluated simultaneously. Compare how the block is used to enable parallelism in lines 30: & 31: with how it is used in lines 46: & 47:.

*** code segment ***

```
35:  /  :: in  component: numerator, denominator
36:        out component: quotient
37:        { quotient = 0;
38:          divisor == 0 $ divide_by_zero ..  ~~error
39:          sign = 1;
40:          -numerator   ${ sign = -sign
41:                          numerator = -numerator
42:                          };
43:          -denominator ${ sign = -sign
44:                          denominator = -denominator
45:                          };
46:          { numerator = numerator - denominator,
47:                 $ quotient = quotient + 1 \     ~~ sequential
48:            };
49:          -sign $ quotient = -quotient
50:          }.
```

Hardware Support

A major motivation for decoupling the language, as has been illustrated by this example, was to permit high performance special purpose hardware to be easily introduced to the architecture in a user transparent manner. A user may define his own comfortable component class and a set of compatible primitive operations. These need not be simple nor disjoint with respect to each other. This done, the

language specification can be used to build and verify hardware.

The core hardware architecture may consist of many asynchronous processors working on different aspects of language interpretation. This permits inexpensive hardware to be exploited via a form of parallelism inherent in the language.

This last feature is rapidly becoming common in contemporary architectures; the contribution made here is that a reasonably high level language is supported as opposed to a virtual memory paging scheme, or a communication protocol, or a graphics standard, or an error detecting and correcting algorithm, and so forth.

Summary

In a normal environment, it is likely that a user would only need to select a component. The system would already know about components and know about vectors. The effort required to obtain a result would be quite a lot less than that expended here.

This discussion did develop five positive and unique characteristics of the proposed language:

1] when parallelism appears in algorithms, it is naturally exploited by D;

2] when algorithms are generic in character, so are their D instantiations;

3] although the basic set of D objects is quite small it can be easily extended, with notational support, to communicate with users on their own terms;

4] the language lends itself to exploiting the capabilities of special purpose dedicated function hardware specified by the user; and last but not least

5] the language lends itself to interpretation by multiple asynchronous independent tasks offering the potential of unleashing the latent capability of emerging microelectronic technologies.

VII. PROGRAMMING SUPPORT REQUIREMENTS

The program development environment currently envisioned to support language users is described in this section. First, the various language formats will be motivated and discussed. Second, development aids will be described. Third, the process of creating a hardware instantiation of a compute bound actor will be elaborated. Finally, some comments will be made about the language in a multiprocessor, multitasking environment.

The purpose of this section is not to specify what should be eventually implemented, but to share ideas.

Formats

A D program, in the course of its existence, is required to exist in three formats. They are the user format, the storage format, and the execution format as depicted in the following figure.

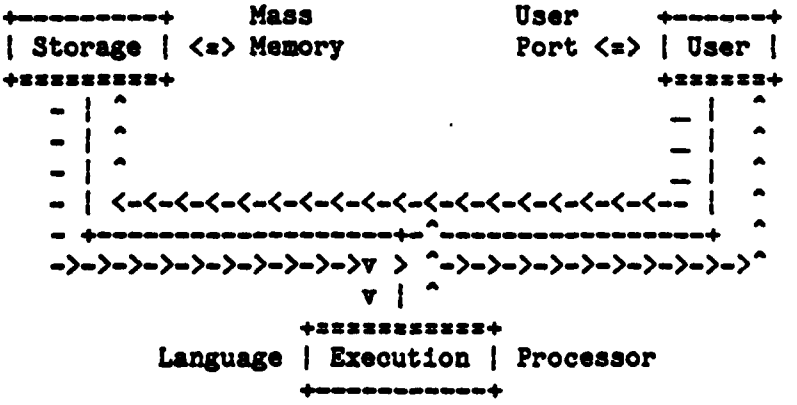


Figure 2: Language Formats.

The most important format is the human readable format, which is the format normally discussed by us (humans). Since the execution unit is not modeled after the human brain, but after the D language, it is

reasonable to expect that a different form might be optimum for execution, and it is. For example, the execution unit does not need multi-character key words, nor object names with many characters, nor (by definition) comments.

The second format is the format in which the program is stored. The motivation for the storage format is two fold: to permit the user format to be reconstructed, comments and all; and to enable efficient generation of the execution format.

At the User Port, which may consist of a video terminal, the user format is translated into the storage format. The language is fragmented, and a single language 'file' is broken into several modules and tables. For example, comments are removed and are replaced with comment markers, a comment table, and a comment file. The comment file contains the comments, the comment table connects the comment markers to the comments, and the comment markers identify where the comments belong in the source file. Similar manipulations are performed on object names.

A source file is listed by the system in cannonical form. Although the system will understand various input formats, it expects the user to adapt to the cannonical output format. Users must learn to appreciate the "free" pretty formatter!

When a program is to be executed, the storage unit sends the requisite files to the execution unit less extraneous markers, tables and files. In the storage format, as opposed to the execution format, object names that the user defines are used to establish a correspondence between files. If two files reference an external object of a particular type named "variable", and if both files are visible in

the same context, storage symbol tables will establish that both names reference the same object by comparing the characters which constitute their names.

In the execution format, the names have been stripped away leaving only tokens which serve as indexes to tables. When files are converted from storage to execution format, linking between the currently active token tables and the files being converted occurs. The figure 3 illustrates these ideas.

The important concepts to grasp are that the Editor and the Linker (which is more akin to the software which implements virtual memory, at the semantic as opposed to binary level) are <u>semantically intelligent</u> and that this intelligence is used to manage the program development environment.

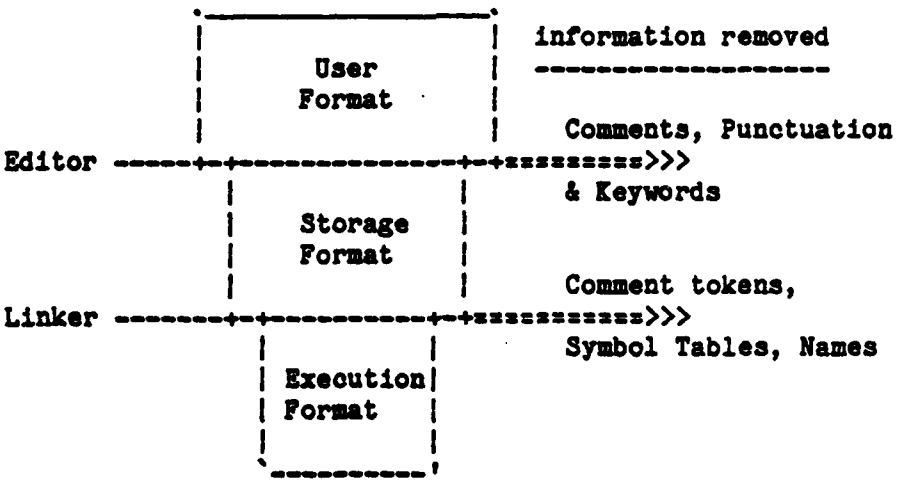


Figure 3: Storage Format Conversion.

A corollary of this observation is that the functions served by the Editor and the Linker are "hardware" in the sense they are immutable in the context of the language.

Development Aids

It is probably optimistic to expect users to sit at terminals and type error free code into the Editor. By forcing the editor to format the source text in a meaningful way, major classes of errors simply cannot be made. It is suggested that in the context of currently popular video screens, the feedback to the user be from the storage representation. Each keystroke must be meaningful. Excellent error diagnostics are also facilitated by these features and should be provided.

Two classes of errors cannot be checked at this stage; naming errors and algorithmic errors. Special consideration should be given to displaying the names of a file, arranged in a meaningful way. Applications which reserve certain names, including operator names, should be able to extend the editor using the modules used to extend the language. Algorithmic errors will require a trace-debug capability described below.

In several instances mention has been made of a macro processor [Cole 81] which would permit the more or less permanent features of the language (punctuation) to be altered to fit specific user requirements. In order to preserve the extensibility sought in the language, this processor must be implemented in the Editor, and must work interactively. Essentially, the semantic gap between the user and storage formats will increase slightly.

If files are to be listed, provisions can be made to recognize the macros. Problems can result from entering a program with one set of macros, and then listing it with another set. If this appears a rare occurance, consider a person who desires to look at the source of system

utility. For this reason, macro processing should only occur during the editing process. Listings will always appear in the cannonical format of the architecture. Another appearance of the "free" pretty formatter.

The trace-debug facility should permit a program to be run interactively, with full access to the runtime structures using character names. The actual runtime structures should be used; not a simulation of them. Although initially increasing the amount of information which must be understood by the programmer, this additional information will permit much more efficient debugging and provide the programmer with a better insight into the logical processes of the architecture.

The last feature that is required to support efficient program development is a library manager. Since the language will fragment big software programs into myriads of tiny ones, automation must be available to manage them all.

Hardware Instantiations

The software tools just described will facilitate the development of actor definitions. A major objective of the architecture is to address applications which are computation limited, and which can absorb the capability of the emerging custom microelectronics capability.

Once a function has been defined, and tested, a special function unit can be designed and tested by simulating the unit with an actor communicating via the accept function and return statement. Such an actor has a scope independent of the calling actors, and can consist of multiple processes, or actors, itself. Each of these may be in any stage of development, definition through hardware.

When a suitable actor has been defined, a detailed functional description of the chip is available at the bit level. In addition, if algorithms are decoupled into control intensive and computationally intensive portions, the computationally intensive portion can be instantiated into hardware. The control portion of the algorithm can take full advantage of the language processor. In most cases this will be significantly faster than implementing the algorithm directly as a single hardware function.

The motivation for removing control is to permit the hardware design to take advantage of regular design structures. This will simplify the design while increasing performance and reliability. Memories, and other arrays of small equivalent functional blocks, are preferred to personal design triumphs. The idea is to keep the user defined hardware simple enough so that a compiler coupled to a generic chip architecture can handle the design.

In a sense, the language defined by this thesis could be considered the functional part of a hardware design language.

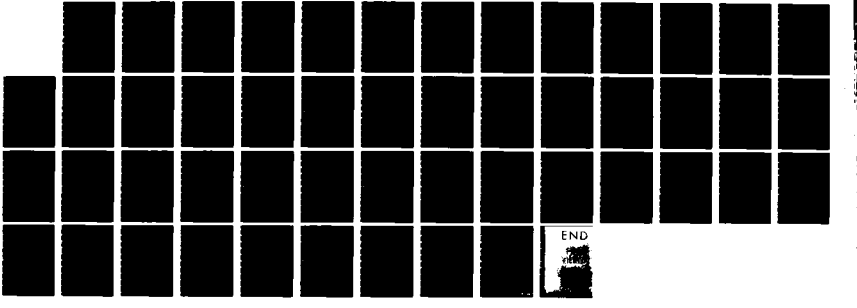
Intertask Communication

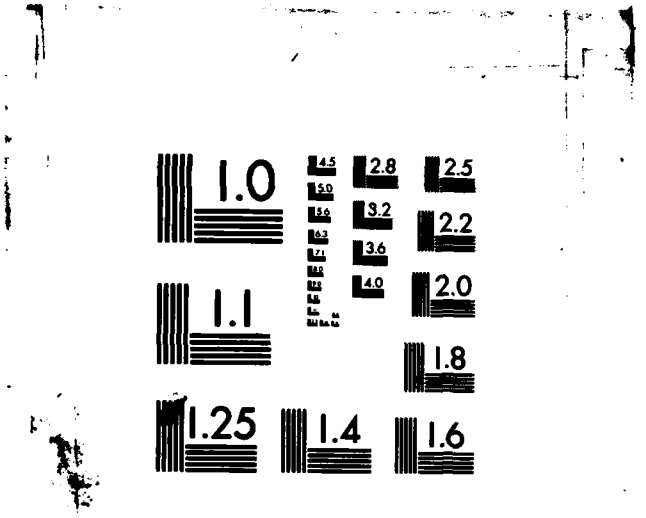
Multitasking within the language is a natural consequence of it, and needs no special discussion. Even single tasks often create many internal tasks which execute simultaneously.

The coordination of multiple processors does introduce some problems not yet described. Considering the most general case of an n-dimensional (n ~ 10,000) net of processors, storage files must be used (in conjunction with some form of capability based addressing) to translate execution formats between processors. The entire system

AD-A138 433 A CANDIDATE PROGRAMMING LANGUAGE(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING R JENNINGS DEC 83 AFIT/GA/EE/83D-1 2/2

UNCLASSIFIED F/G 9/2 NL





cannot have a single token table for all objects. Each processing node must maintain its own symbol table.

The Bus Interface Unit is given the responsibility of maintaining the files in their pseudo-storage format, and performing the conversion to execution format on the fly. A generic processor is illustrated in the following figure.

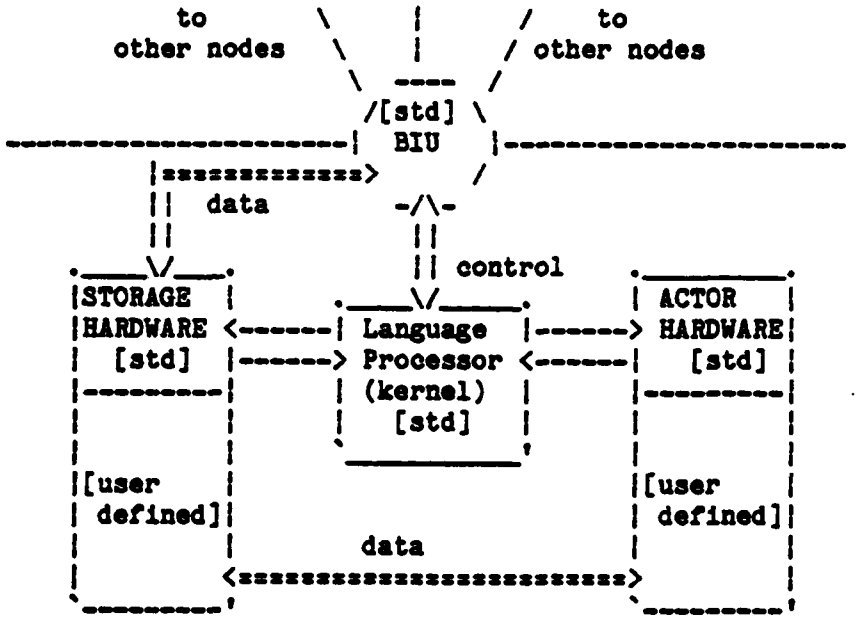


Figure 4: A Multiprocessing Node.

Summary

The important message of this section is that the task of presenting a consistent, comprehensible, and complete view of the internal structure of the proposed architecture to users forms an implicit constraint upon its design. The method of providing programming support to the user has been, and continues to be, an integral part of defining the architecture and the language.

VIII. HARDWARE REQUIREMENTS

Even though it has been argued in this thesis that hardware should be designed to support the language, the capabilities of hardware have significantly motivated major portions of the language. To insure that hardware can efficiently support the language, it is imperative to define a language which can be so supported. The purpose of this section is to motivate three portions of the language which were driven by hardware considerations.

The utility of this section to the thesis is threefold: 1) it provides rationale for what might otherwise seem esoteric syntax; 2) it provides insight into how the language is put together; and perhaps most important 3) it should serve to mitigate (if only slightly) exclamations of "it can't be implemented!"

The three basic hardware related problems which were encountered, and will be described in this section, are 1) defining an application independent kernal to interpret the language, 2) developing an efficient control structure, and 3) catalyzing the exploitation of special purpose hardware constructed from Very Large Scale Integrated (VLSI, spelled VHSIC by the DoD) circuits. In each of the following three cases the problem will first be summarized, and then the proposed implementation model will be described. While this section will fall considerably short of defining the structure of a D machine, it should provide the major insights required to attempt a D machine design.

Problem 1: Kernal Definition

Conventional computers achieve their general utility by including huge numbers of instructions, occupying microcode or incorporating logic that is seldom used. The Reduced Instruction Set Computer (RISC) developed at UC at Berkeley [Patterson 81] has focused on this inefficiency, and now a commercial product is available which has a limited instruction set and extra general purpose registers. It is claimed to be twice as quick executing PASCAL and C programs [Morrow 83] as "traditional 32-bit computers".

Why not use this as a kernal? The objectives of this thesis are not met because of the limitations of C and PASCAL. For examples, type independent algorithms and parallelism cannot be adequately supported. In retrospect it is obvious that if it is known, apriori, that only C and PASCAL will be used on a machine, and that C and PASCAL require only a subset of the machine's capability, a simpler machine could be used to support C and PASCAL. In the context of a given technology simpler machines are faster machines.

The objective here is to move in the other direction; to unconstrain programmers and let hardware support what ever they want, while at the same time unburdening them of excess complexity. The excess complexity required to meet the requirements of some other application which must be supported to sell the requisite number of machines to justify development and production tooling costs.

The desire is to provide the smallest common denominator each user can live with. In the context of the utility measure discussed in section II of this thesis the smaller the denominator is, the greater the incentive will be for widely adopting it.

Expression Evaluation

The basic purpose of any algorithm is to manipulate values of objects. These manipulations are inherently application dependent. The approach taken toward finding the smallest common denominator for all people who want to develop and automate algorithms was to develop a language which would permit the user definition of tokens within an expression. In addition to serving users better, this saved a lot of effort trying to guess what such a set of universally useful tokens might be, and then developing them to provide to users as a language predefined set.

With great humility it was realized that language users, irrespective of the language, generally know more about the structures they need to support their application, than do the original language designers at the time the language was designed.

This decision made, the language can be partitioned (as it has been) into a part common to all language users, and a part specific to a particular application. The common part consists of defining the tokens which constitute expressions (static object names, literals, operators and functions), and the structure required to meld the expressions into meaningful algorithms incorporating conditional execution and dynamic context management. This part is the part of the language which all users must use to describe the algorithm they want to implement, and the abstract concepts they would like to use within expressions.

The second part is expression evaluation, which is almost purely subject to the needs and requirements of the user. Before the expression evaluation is described, it should be noted that two questions arise: 1) what should the control structure look like? and 2)

how should operators and functions be modeled so as to permit the most common functions of an application to run fastest with VLSI support?

The following figure illustrates how a subexpression, A*B, is evaluated and assigned to the temporary variable C. To the common part of the language, a token representing an expression is simply a token representing a value. In some cases, for example a nominal guard evaluation, the value must be converted to an integer before it is used.

To obtain the value, the language model requires that the expression list, consisting of tokens representing actors and static objects, be placed on the evaluation stack. In the figure, two versions of this stack are presented. The initial stack shows the name tokens (*, A, B) which are on the stack at the particular moment we begin to watch. The final stack shows the temporary token (C) which holds the value produced by computing A*B. The action to be described is A * B => C in the midst of a larger expression involving the name tokens (A again, *, G, H) and more. A name token does not refer to an object, it is just shorthand for a particular name. Consequently, it is not associated with a name or a class.

What must occur first is that a subexpression must be recognized to be ready for evaluation. In this case, * is recognized to require two arguments, of a particular class, which A and B are. If A or B were different classes, then activation of * would be preceded by a class conversion. Since everything is ready, a transaction identification number, <trans_id>, is assigned to the operation, and an internal object name is created, C, and a result token, <r_token>, is created for the result.

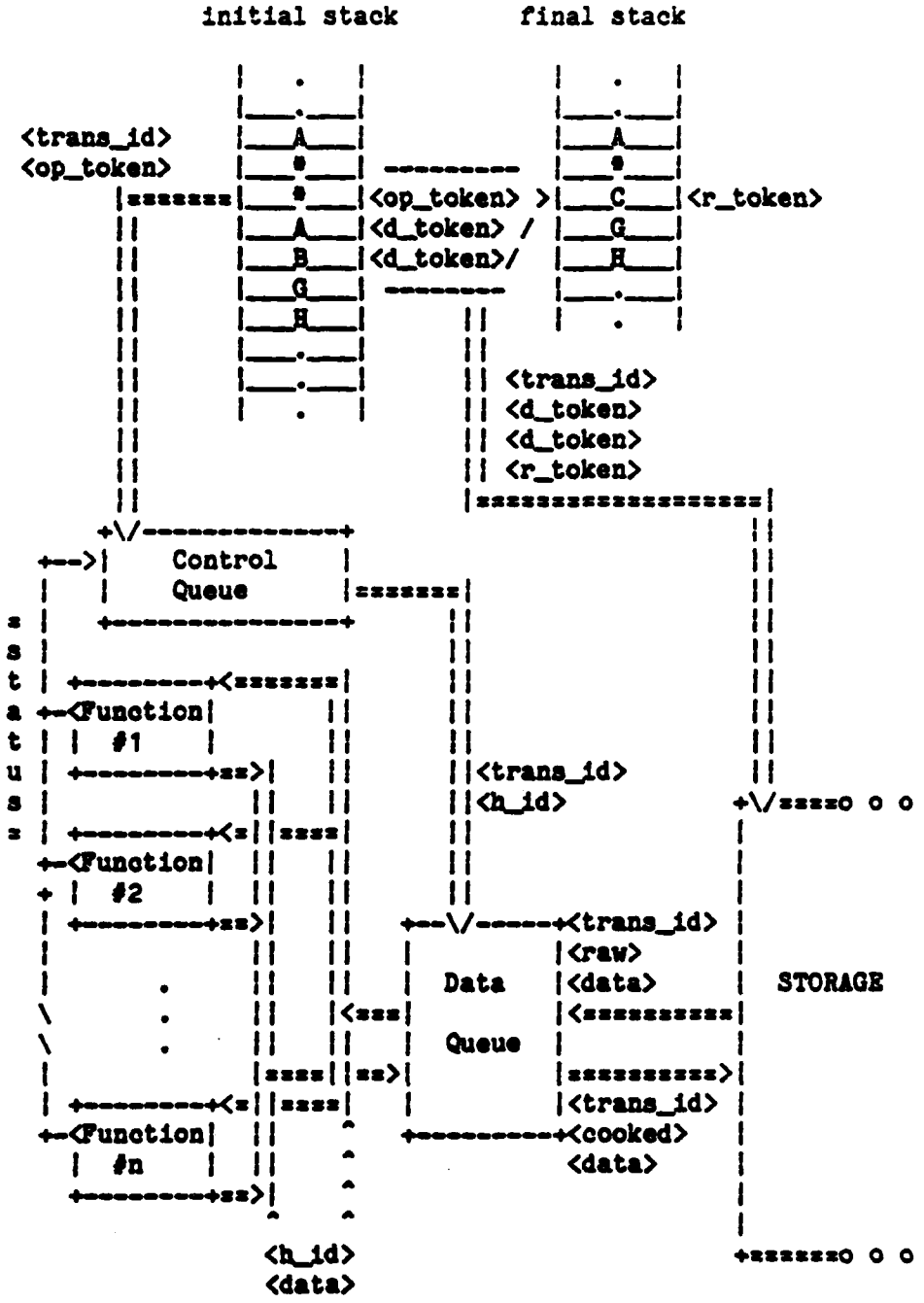


Figure 5: Expression Evaluation.

A message is sent to STORAGE where the values of all the objects are held which consists of <trans_id>, the two <d_token>'s, and the <r_token>. In other words: fetch the value associated with the <d_tokens> and put them on the Data Queue to be referenced by <trans_id>. After this is done, forget about the <d_tokens>. When the result comes back, assign its value to some temporary object which will be referenced in the future by <r_token>.

When STORAGE can produce results for the <d_value>s they, with <trans_id>, will be sent to the Data Queue.

Simultaneously <trans_id> is sent to the Control Queue with the <op_token>. What occurs here is essentially the problem alluded to earlier. As will be described in a following subsection, a dynamic symbol table links the <op_token> to, in this case, a hardware special function unit which is identified by a hardware id, <h_id>. This may not occur immediately, but will occur when the hardware unit is available, and <trans_id> has the highest priority of all the waiting transactions.

When the hardware unit is allocated by the Control Queue, <trans_id> and the just identified <h_id> are sent from the Control Queue to the Data Queue. This enables the Data Queue to put the data it associates with the transaction on the proper bus and to wait for a result.

When the hardware unit is done, it sends the result back to the Data Queue and a ready signal to the Control Queue. The Data Queue attaches the result data to <trans_id> and sends it to STORAGE. If STORAGE has something else to do with it, it is routed back to the Data Queue as data in another transaction. If not it is stored, awaiting reference, accessible via the token <r_value>.

Three important things to understand about this scheme are: 1) it maintains the partition between the common part of the language and the user definable part; 2) it permits subexpressions to be executed in parallel, but in sequence when required; and 3) it breaks up the expression evaluation process into many smaller processes which asynchronously communicate with each other.

Problem 2: Structured Control

There has been much debate about whether goto statements are good or bad. People who avoid them claim, and rightly, that they can be used to write code that is impenetrable. Why, one wonders, is this capability a requirement for contemporary programming languages? The answer is that conventional structures are not powerful enough, or at least they are not elegant enough.

Although it is not widely publicized, some programmers (at least one) are also frustrated with the if-then-else construct which appears benign enough in programming texts. Out of the isolation of a trivial example, with five to six (or more) of its esteemed colleagues, one is faced with, again, impenetrable code. Most language designers have attempted to mitigate programmers' frustration with conventional control structures by providing several different structures, often blending in iteration control as a lagniappe.

The objective is to find a comprehensible way to tackle iteration and conditional execution, and to do so in a way that it could be implemented efficiently (using the smallest address space possible), in the context of the solution to the previous problem.

Block Execution

The approach taken was to extend the properties of the "block" in two significant ways. A traditional block is a series of statements which are executed sequentially, and each block may modify its scope by introducing local variables. A D block can be thought of in a similar way, with each statement 1) preceded by a guard expression and 2) followed by a continuation.

This has the effect of making each statement of the block a conditional statement (covering conditional execution), and a "goto" statement with a limited jump capability: control may be passed to the beginning of the block, the end of the block, the next statement (regardless of the guard value), or the next statement with an open guard. Blocks can be labeled, and the labels used to place the jump commands in the context of any enclosing block.

In contrast to the traditional panoply of control statements, this extended block structure is simpler to understand and to implement --- despite being considerably more powerful. The implementation model is depicted in the following figure, and consists of a block stack, and a block table.

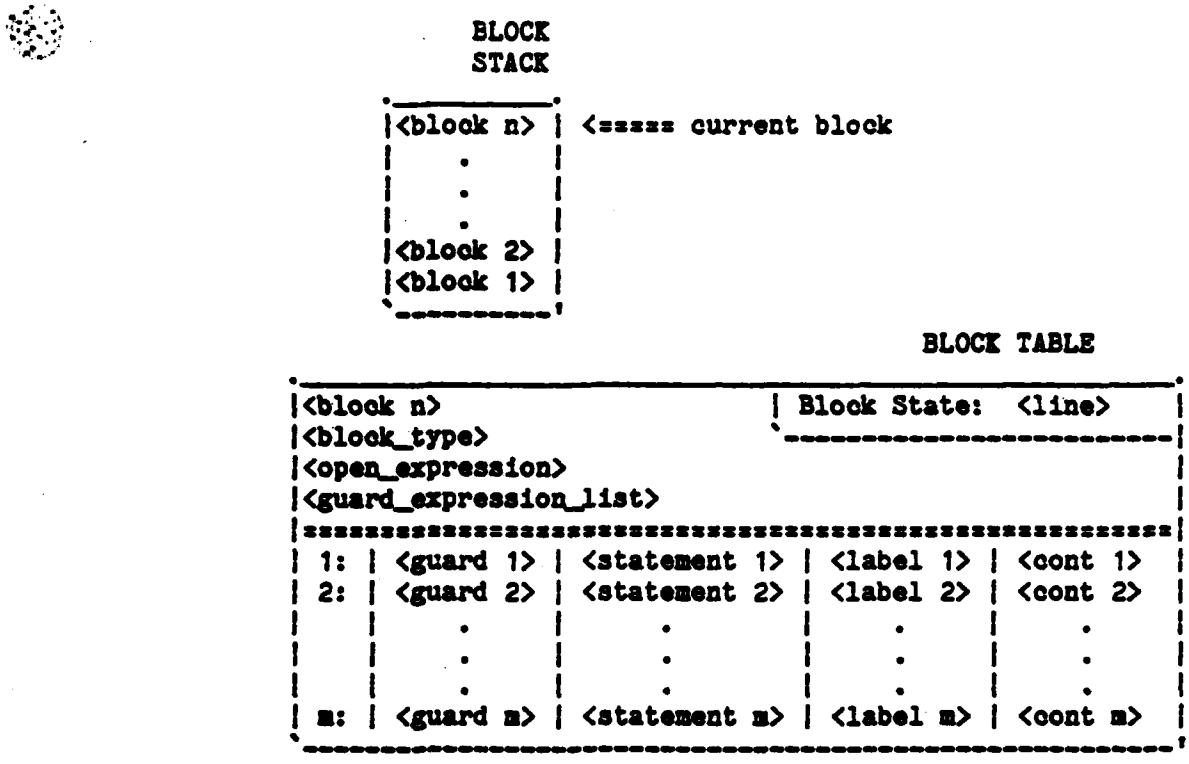


Figure 6: Block Execution.

The Block Stack keeps track of the current block which is active. Within a task, or single invocation of an actor, only one block can be active at a time. When a child block is entered from a parent, the child block's identification is pushed onto the Block Stack and becomes active. The parent's Block Table is saved until the child block is done. When a Block Table is saved, the current line number serves to continue the block state when the Block Table is recalled. In the figure, the current block is identified by the token <block n>, which appears both in the Block Stack and on the current Block Table.

The block type influences how the guard vector of expressions and values is treated. In any case upon entry into a block the open expression is sent out for evaluation followed by the list of guard expressions. If an element of the guard list returns a value equal to

that returned by the open expression, a value of open is entered in the Block Table.

Normally, (no (?) appearing in the user format), each guard value is considered in list order to determine if control should pass to its following statement. If it should, control passes to that statement (even if all the guard vectors have not been evaluated); otherwise it waits for guard on the next line to be evaluated. The block cannot be left until all guard expressions have been evaluated.

The other type of block, (a (?) following the block label), passes control to the first guard, regardless of its position on the list, which returns an open value. Should the block end, all pending guard evaluations are flushed. If the block is saved, guard evaluations continue.

In the Block Table, a guard has the value of open, closed, or unknown. Each statement essentially consists of a block token, indicating entry to another block, or an expression token. All statements, with the exception of blocks, can be reduced to expressions with language defined functions.

The label field contains an optional block identification. The continue field contains a block label and a token which determine the thread of control.

Unless the statement consists of a block, the only action taken by the language processor is to put an expression token onto a queue for execution. In a compute bound system, one such processor can handle several tasks. After putting the guard list of the first task on the execution queue, it can start on the second task's guard list and so on. When a guard from a higher priority task returns an open value, the

processor saves the current task upon which it is working and spools expression tokens onto the execution queue from the first task until it must wait for another guard to be evaluated.

It should be apparent that execution time is dependent upon how fast special purpose hardware can perform its function; although this language does require overhead it does not effect, to the first approximation, elapsed time required for execution.

Problem 3: VLSI Exploitation

A natural consequence of most "extensible" languages is that no matter how well the new functions can be implemented into the language, unless the hardware supports the new structures efficiently, slow execution speed precludes widespread acceptance. The emerging performance afforded by custom VLSI [Foster 80] at affordable cost offers an opportunity for users to customize their hardware configuration much as they have customized their software. In stead of forcing users to purchase entire machines to obtain quick execution of just one algorithm, for example a Fourier Transform which is both generic and has a wide enough user base to justify the development of an entire processing system just to support it, the idea is to permit users a path to a quick (VLSI hardware) version of one of their own compute bound operators or functions.

What is sought is akin to a switch on an "optimizing" complier which says: optimize the hardware as well as the software. Contemporary architectures are a long way from this as evidenced by the dearth of software which is able to exploit the plethora of fantastic hardware available to augment the IBM personal computer, and its clones.

The problem is epitomized by what it requires to get standard compilers to exploit the capabilities of the 8087 (floating point chip) which was designed into the IBM-PC, simply to speed up a common function.

Actor Execution

In D a standard linkage convention is imposed for actors which are supported by hardware units, for actors which are supported by accept statements, and for those actors which have only been specified within a definition file. The following figure illustrates the processes which contribute to the meaning of a transaction. A transaction identification token, <trans_id>, is sent with an actor token, <act_token>, to the Control Queue. An <act_token> can be either an <op_token>, as it was in the previous discussion of expression evaluation, or it may represent a function token.

An actor token can reference a hardware unit, a task hung on an accept function, or an actor definition. An actor instantiation must be visible at the time it is invoked. Hardware hides waiting tasks, which hide definitions. Linkage can be used to override this architectural bias.

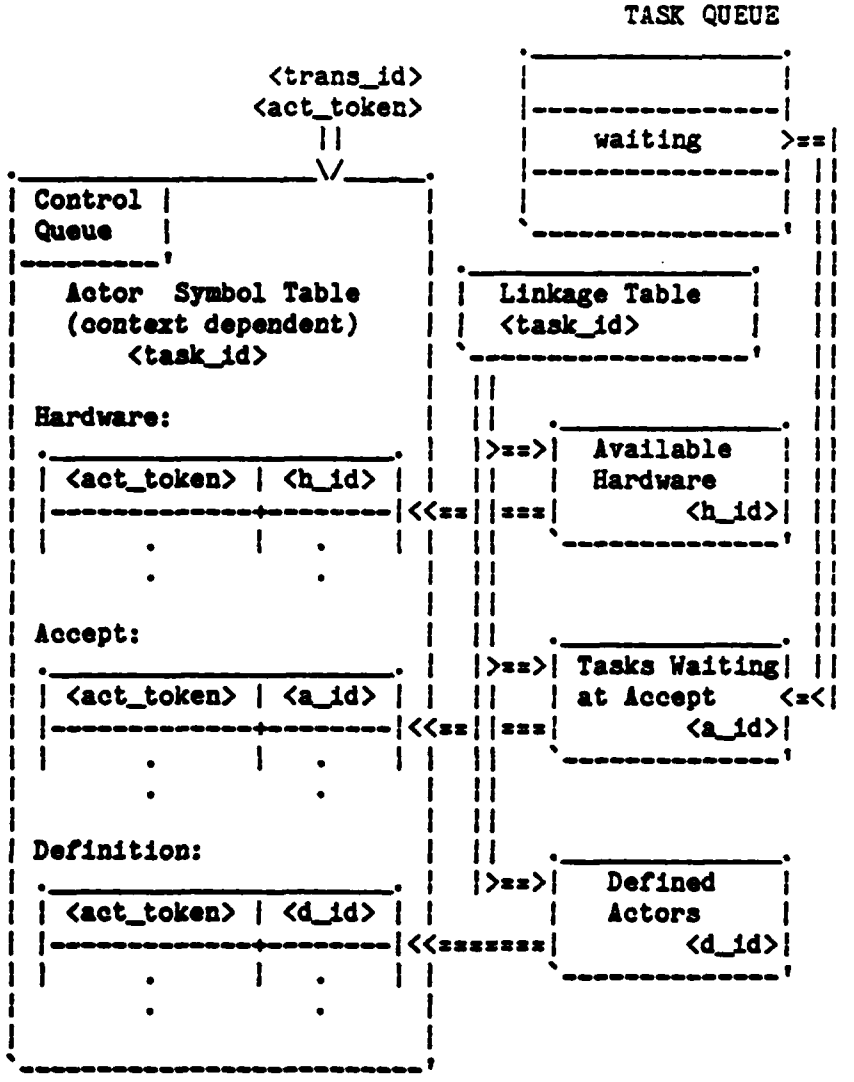


Figure 7: Actor Execution.

Each task has its own linkage table which is used to create three tables, within the Control Queue, of resources which are available to execute operators. These tables are changed as blocks are entered and exited, as tasks accept function calls from other tasks (in a multitasking environment), and as hardware becomes faulty, and then again when the hardware is replaced.

If a hardware unit is available, as described above for expression evaluation, the <act_token> is replaced by a <h_id>. Similarly, if the

operator is to be connected to a waiting accept then the <act_token> is replaced by an <a_id> which the Data Queue is smart enough to recognize. Instead of sending data to the hardware units, it sends the data to the Task Queue which returns the appropriate result. Should it be required to invoke a definition to obtain an evaluation, then a <d_id> is routed to the Task Queue via the Data Queue which causes a new internal task to be created which will live until the actor result is returned. In this latter case, a mechanism for inducing parallelism has been described. Within an expression, each function may define an internal task, and may be simultaneously active.

In all cases the computed result is returned to the Data Queue, and subsequent activity parallels that described for expression evaluation.

Summary

The D programming model roughly consists of three basic ideas touched upon in this section. Each idea can be thought of as an independent process which communicates asynchronously with the other two.

The first works its way through code, evaluating some expressions to determine sequencing, and just dumping others to be executed on expression queues and stacks.

The second takes the expression stacks, and compresses the expressions to values, making any assignments to permanent variables as required.

The third is a giant dynamic symbol table that ensures that the various object tokens, which the various processors use to reference objects instead of names, always reference the proper objects

consistently with respect to the user program. It is acknowledged that the third example examined only a small, but representative, piece of this problem.

IX. LANGUAGE COMPARISON

Since an implementation has not been accomplished (yet) some form of analysis should be performed to determine if the proposed language is workable. The approach selected is to provide, in this section, a comparison with the C programming language [Kernigan 78].

The objective is to provide a basis for the suggestion that the proposed language is both sufficient to replace the C programming language, yet necessary to efficiently program future systems. This will be done heuristically with the intent to convince the reader that the proposed language is as capable as C.

The D language will be discussed in the context of the C language, following the general format of "The C Reference Manual" as it appears in Appendix A of The C Programming Language [Kernigan 78]. The numbers provided refer to the sections of "The C Reference Manual". A basic understanding of C is assumed. To completely understand this comparison D should be understood as well. The problem with completely describing D in this format is that while the format is fine for C, D is a different language which is more lucidly covered by approaching its capabilities from a different perspective.

By selecting this approach, as opposed to a rigorous "proof", C programmers also obtain a quick introduction to the D language in familiar terms. At the time of this writing, a D environment does not exist. Consequently, a viable approach to learning D is through a thorough understanding of C. Computer architectures in general use change very slowly, and C is unquestionably the language of choice for current architectures --- so time so spent will certainly not be wasted.

A rigorous proof should be attempted using the concept of structural induction described by Stoy [Stoy 77], and is left as an exercise for the determined reader.

Introduction (1)

The C language was primarily developed for the PDP-11 architecture. Although it has been ported to many other commercial architectures, and been ported relatively efficiently, this should not be interpreted to mean that it can be efficiently ported to all future architectures. It should be interpreted to mean that most of the architectures in use today are similar to the PDP-11 model.

For such architectures it makes obvious sense to constrain the extensibility of the language to prevent the programmer from creating complex and inefficient programs. For this reason, C has remained simple, and is now the language of choice for many programming chores in 1983.

New architectures are not so constrained. The trend is to exploit special purpose processors for specialized functions. Consequently, if specialized functions can be isolated and defined, they can be instantiated in hardware. New architectures, including the architecture suggested by D, supports this approach.

D contributes structured extensibility: the language structures the user extensions as it does the language kernel. C can also be extended (which is one of its major strengths) by defining types, declaring functions, and using the macro-processor included with most C compilers. The extensions are by definition inefficient since they must be implemented though C primitives, and do not always benefit from the underlying structure of the C language.

For example, if one is not happy with the implementation of floating point in C, short of rewriting the compiler not much can be done. For those who claim that all arithmetic can be accomplished by using function calls, it is suggested that users of such an agglomeration are not fully benefiting from C structure. They are not, in a pure sense, programming in C but in some bastardized functional language of their own design.

In short, C makes a great many more assumptions about the architecture of the underlying hardware than is required. In the context of Very Large Scale Integrated (VLSI) circuits, D provides an alternative which is developed below.

Lexical conventions (2)

Comments in D are terminated with lines (by line terminators), and consequently are more limited than those provided in C. As will be argued in many cases below, a macro processor can easily extend the features provided in D sufficiently.

Identifiers may be of infinite length, limited only by the line length and comprehensibility.

There are 13 reserved keywords in D as opposed to 29 in C.

Literals serve the role of constants. They are names which start with a digit or underscore, and may take on values only by initialization. Digits are predefined literals. Characters are predefined and may be manipulated via quoted literals, which follow the C convention. The backslash convention is not implemented as it can be left to a macro processor.

Syntax Notation (3)

The syntax is summarized in the appendix. The notation is described at the beginning of section V entitled "Language Description".

What's In a name? (4)

In D the lifetime of each object is indefinite. The storage unit for every variable is a file which continues its existence as long as it is referenced. Within a actor definition, no variables are locally declared, so they all are "external" in the C nomenclature.

Automatic variables are limited to the arguments and results of functions in D. Register variables have no meaning because the D programmer views an unlimited number of virtual registers which contain user defined objects, not some arbitrary number of binary bits. To make an object static, it only needs to be declared in a file that is put into a system directory. Until its referencing file is removed from the directory, or the system is terminated, the variable will remain.

D supports exactly four types of basic objects: scalars (which can be mapped onto a finite set of whole numbers); structures (which are composites of scalar and structure types); functions; and operators. All other types must be user defined. This does not mean that each programmer must write his own floating point package, only that the decisions made in designing and developing a floating point package have less to do with this language than most applications.

Classes, similar to C types, may be modified in three other ways:

1] they may be changed into an array of objects of a class;

2] they may be changed into a pointer to an object of a class; or

3] they may be made generic; e.g. the class of an actor's formal argument is deferred until the actor is invoked.

There is no requirement in D for unions since they are motivated by PDP-11 architectural limitations ... PDP-11's think in terms of bytes, and worry about word alignments.

Objects and lvalues (5)

Objects and lvalues (short for location of values) are common to both D and C.

Conversions (6)

The concept of conversions in D is in concept similar to the approach taken in C, but more abstract, again because of fewer architectural assumptions in D. The rule is that implicit conversions, which may be required to evaluate an expression, may always be made from a source class to a destination class if 1) a conversion function exists and 2) the destination class has a greater dynamic range than the source class. A greater dynamic range is assumed to be equivalent to requiring a larger block of memory for each object. In C this would be equivalent to permitting implicit conversions if

sizeof(<destination_class>) > sizeof(<source_class>)

Pointers and integers cannot be mixed in D, because each operator can only accept operands of a single class. If a programmer wishes to increment a pointer by n objects, n is converted to a pointer value, and then the pointer is incremented. This done by the implementation implicitly. Pointers are intimately tied to arrays in D as they are in C.

Expressions (7)

Precedence of operators is determined by the right-most operator symbol of their name. In the BNF description of the operator symbols, the first two symbols have the highest precedence, then the next two, and on to the lonely '=' which has the least precedence of all. This list represents a modifiable table which should be, not easily, user accessible in an implementation defined manner.

The order of subexpression evaluation is undefined, and may be different upon subsequent executions of an expression since it is determined at runtime. Users define how division by zero and overflows are handled.

A primary expression is called a <lname> (logical name) in D, which can result in either an address (location name) on the left side of an equal sign, or a value on the right side. Function calls, class modification symbols (# = * in C, @ = & in C, [], $) group from left to right, as does the component selection operator ('), which operates on structures.

There is no equivalent (->) operator, which combines the pointer and selection operations, since the renaming capability serves much the same need.

No implicit conversions are made. A name declared to reference a function will never reference a pointer to that function. An array name without brackets is the <lname> of an array.

Functions are called with only one argument, which may be a structured object. The argument may be a pointer to the composite or the composite itself.

No types are implicitly converted unless the conversion is required for an expression to be evaluated, and then conversions which cause the least growth of the size of the destination class are attempted first.

All operators except (+, -), acting upon integers, must be defined in the language by the users. The assignment operator is predefined for all user defined types, and does perform full class checking. The language can easily support more complex hardwired functions and operators, but these must be described within D. With such a description, should the hardware fail, the function would still be available in terms of the basic hardware although it would take longer to execute.

The tertiary conditional operator has no equivalent in D, and was deemed redundant in the context of the other conditional execution structures.

The comma operator would serve no purpose in D, and no consideration was given to implementing it.

Declarations (8)

Declarations in D always reserve storage, although not immediately. Storage is reserved when the storage declaration file containing it is copied, and when a storage file is given to the system by the editor. Components declared within structure definitions may only reserve storage through storage files. Declarations of formal in or out variables in actors do not reserve storage until the actor is activated.

Each declaration may contain an initializer, in fact it must if a literal (constant in C) is being declared. Structures and arrays may only be initialized by one object -- a structure or an array of the same

class. Such an initializing expression must be composed from available operators and predefined objects.

Class modification tokens (#, []) must occur juxtaposed the name of the declared object, while the ($) token is applied to a visible object name abstracting its class.

While a C declaration can permit multiple arrays and levels of indirection to be declared at once, permitting parentheses to define precedence, D requires that one level of indirection be declared at a time. In order to efficiently, reliably and unambiguously track multiple levels of indirection and arrays there appears no better way. What this means is that what might be declared in C to be:

```
int i, **ptr;
=>      i = **ptr;     /* two levels of indirection */
```

in D would appear

```
int: i, #iptr
=>      @i = #ptr = iptr          -- one level

$iptr: #ptr
=>      i = ##ptr = #iptr         -- two levels
```

In the C scheme, there is an implied pointer which is not available to the programmer. The D scheme makes this pointer explicit. This elaboration also is applicable to arrays.

Bit fields are another anachronism tied to the PDP-11 architecture omitted from D. To achieve the same effect, scalar classes can be declared with the requisite number of values, and then packed into a structure.

Similarly unions and the sizeof construction, necessitated by the PDP-11 architecture in C, contribute nothing to the D environment, and have been omitted.

Parentheses within declarators are not permitted; to create multiple levels of indirection and or arrays, multiple declarations are required. Since C does not incorporate strict type checking, parentheses can be used in C to approximate these effects. In D the implicit intermediate variables must be explicitly declared.

The state definition file performs all the class definitions. In D, the user I/O format of an object is tied to the object, and is not arbitrated by a format statement. The effect of a format statement may be achieved by performing a class conversion before printing.

In C the memory format of the different types is described by the implementation documentation, and can be exploited by the so motivated programmer to achieve must the same effect with a library of conversion routines. In D, the language and the implementation are much closer, so hooks must be explicitly put into the language to support such flexibility.

In structure definitions, component initializations may be performed. These will of course be overridden by declaration time initializations.

Statements (9)

Statements are generally executed in sequence in D as they are in C. The sequence is controlled by the structure of the compound statement or block, and not with a set of conditional statements.

Each statement is imbedded in a line, preceded by a guard expression and followed by a continuation. The continuation serves to send control back to the beginning of the block, or any enclosing block, (similar to the C continue); to the end of the block, or any enclosing

block, (similar to the C break); to the next statement regardless of the guard value; or to the next statement with an open guard. All guards are evaluated at the top of the block, and they are open if they evaluate to a positive value, or the value provided by the optional selector.

Each block is similar to a C switch, where the switch argument and each of the case values can be expressions (guards). There are no goto statements within the language, and all sequencing must conform to this model. Each such block is labeled, so the continuation can be put into the context of a particular enclosing block. This is equivalent to a compound C statement or a C block, with bells and whistles.

The power of the D notation can be seen when it is compared with the C conditional statements. All of the capability, and then some, has been integrated into one efficiently implementable structure.

The other three statements are essentially expression statements. The first is analogous to the expression statement in C.

Table II: Control Statement Comparison: C vs D

C Statement	D Equivalent
if (<expr>) <stmt1>; else <stmt2>;	{ <expr>$ <stmt1>.. <stmt2>.. }
while (<expr>) <stmt>;	{ <expr>$ <stmt>\ }
do <stmt> while <expr>;	{ <stmt>; <expr>$\ }

switch (<expr>) {
case const1: <stmt1>;
case const2: <stmt2>;
default : <stmt>;
}
	D: <expr>
	{ const1$ <stmt1>
	const2$ <stmt2>
	<stmt>
	}

The second is called a return statement, but functions slightly differently than the C return. Since the result object is explicitly declared for each D actor, the statement "return <expr>" assigns the value of <expr> to the result object, and then executes a actor termination returning control to the calling actor. If the actor call was serviced by an accept, the servicing actor is not terminated.

The third incorporates the alloc function of C into the D language with a keyword of its own. The statement "new <pointer_name> = <expr>" creates a new object of the type <pointer_name> points to, assigns <pointer_name> to its address, and initializes the object to the value of <expr>. To dispose of an object, all references to it must be eliminated. Immediately after a declaration, "<pointer_name> = null" would dispose of the new object.

The latter two statements are considered language defined functions augmenting user defined expressions. Neither function returns a result to the invoking task.

External Definitions (10)

In C all files are more or less created equal. In D there are four types of files: files in which passive types such as integers and structures are defined, files in which active types such as functions and operators are defined, and files in which all types of objects are declared, and which are responsible for allocating storage. Then there are files which essentially serve as libraries of other files which are related in some way, and which are usually referenced together.

In most cases, D objects are defined and declared externally. That is, one file established a link between a name and a storage location (declaration), another file determines how to interpret the value stored (state definition), and yet a third file determines how actors manipulate static objects -- or how names are manipulated within an algorithm. In C, most objects are either declared in the same file they are used, or labeled extern causing the compiler to link the name with a declaration in another file. The concept is the same, but D has extended and regularized it.

For a particular algorithm, all the state definitions need not be in one file, nor the actor definitions, nor the declarations -- hence the motivation for the library, or bundle, files to organize the otherwise intractable plethora of little files.

Much of the power of D is a consequence of the ability of the links between these files to be forged at what conventionally is called "runtime". The modularity which provides C with much of its capability to support large software projects has been conceptually extended to a higher level of abstraction in D, permitting user defined objects to enjoy the support of language defined objects: free use of operators,

simplified I/O, and object transparent actors (algorithms written in terms of formal actors and formal passive objects which are object independent); to name three important features.

Scope Rules (11)

Lexical scope rules are similar to those of C, except that object declarations are introduced indirectly by linking to declaration files, or bundles naming declaration files.

As previously mentioned, most object names are by default extern in the C sense, the exceptions being definitions defined and used in the same file, and initialization expressions composed of variables also declared and initialized in the same declaration file.

Privacy from other files can be obtained by copying, which obtains a private copy of the file and all the objects declared within it. If the variable is subsequently referenced by an object in a file disjoint from both the declaration file and the acting file, its life becomes independent of the two. This construction meets the needs served by C's static variables.

The alternative method of obtaining access, sharing, serves the purpose of of providing a communal set of objects accessible to several functions. This meets the needs served by C's external variables.

Compiler Control Lines (12)

Comments have been made in several instances that features of D have been omitted because, if desired, they could be implemented in a macroprocessor. Compiler control lines are an obvious example of such features, and are deemed to be outside the scope of the D language. This does not mean that such a tool would not be part of the D

development environment, only that it is not part of the language.

An accurate analogy would be between the instruction set of a microprocessor and the capabilities of the software designed to support program development upon it.

Implicit Declarations (13)

There are no implicit declarations in D.

Types Revisited (14)

Unlike C, structures in D are equivalent to other types and no limitations of any kind are imposed upon them.

The implementation hints provided in this section (of the "C Reference Manual") are not relevant to the D programming model because D does not sacrifice conceptual clarity for implementation efficiency. The motivation is to use VLSI to pay for a conceptually elegant programming model.

Constant Expressions (15)

The concept of a constant expression arises when the compiler must be able to determine values of expressions. In D this is not a problem since all expression evaluation occurs at runtime.

Portability considerations (16)

Historically, languages have tended to have a much longer lifetime than computer architectures. The philosophy motivating D has been to let the architecture of the language drive the hardware (within the limits of VLSI). Consequently, by design, no consideration was given to portability of D to conventional architectures. Such an exercise is left as an exercise for the reader, to be accomplished after using structural

induction to formally prove that D is both necessary and sufficient in the context of a conventional architecture. (It is admitted that "the ice is thin here".)

Anachronisms (17)

To be determined by future users.

Syntax Summary (18)

The syntax summary provided in the appendix of this thesis should provide a basis, with very few exceptions, for implementing the language.

Features not included in C

Learning should be in the context of computer programming, and it should not impose a burden. The language itself should facilitate the organization and comprehension of concepts and data.

The flaws in C, and in all of the widely used "higher order languages", are that only limited abstract concepts, types, are effectively supported, and generic algorithms are not supported at all. Similarly, there is no support within the language for exploiting special hardware. While C aficionados may produce code which supports either of these objectives, they are in effect programming outside the language.

D does explicitly support generic functions and operators. In fact, because passive and dynamic objects are defined in separate files, all algorithms are basically generic. This is not a consequence of an added feature, but is a characteristic of the D language model. The apparent user benefit is that user defined types and operators are as

easy to use, in terms of efficiency and syntax, as those built into C. In addition, special purpose VLSI hardwired functions can be easily linked to user defined operators.

The second major extension concerns real time support beyond bit manipulation. The basic control structure has a real time flavor in which the guard vector may consist of expressions containing the language defined accept function. Control is passed to the statement following the first (in time) open guard, regardless of its position in the block. While this structure might not always minimize interrupt response time while a background task is executing, it is fast, it is reliable, and it is a structure very much a part of the language model.

Summary

At this point it should be apparent that D is a functional superset of C, and that there are solid reasons for seeking to extend C. Although C is a widely used language which currently defines the contemporary programming environment (circa 1984), it should be clear from this section that future technologies currently becoming available to implement computer architectures permit beneficial extensions to C, as described above, to be envisaged.

VII. CONCLUSIONS

The basic motivation for this thesis has been to explore the optimality of current information processing systems in the context of current processing needs and current implementation technologies. This has been accomplished by proposing a superior method of employing automation.

Three topics need to be discussed to conclude this discussion: 1) what is the best path to a functional demonstration of the proposed system; 2) what other work is required before the full potential of the architecture is realized; and 3) what is the current motivation to continue with the effort.

Functional Demonstration

The user format of the language has been defined, and is described in this thesis. The storage and execution formats have not yet been defined, and must be before the language and hardware are finally defined. These tasks consist of creating tables to represent the four file types in storage, and creating transformations which represent action in the language. Generating these tables may require a final iteration of the language, and will suggest a complete hardware configuration.

This task may be facilitated by creating a D compiler and support tools for a conventional architecture, and then using this software for simulation and testing. Another alternative would be to breadboard, with TTL and microprocessors, a kernel with hooks onto a standard bus (e.g. S-100, Multibus, Q-bus, or the IBM-PC bus).

Using a conventional microcomputer system to handle the basic housekeeping chores, and a special purpose hardware unit to interpret the language and interface with some representative special purpose hardware would maximize the utility function for the greatest number of potential users. The language would be executed quickly, and the incremental cost of a language system would be low.

Future Work

The reliability of the architecture needs to be put on a firm basis, and methods of object referencing need to be developed.

Reliability can only be assured if a firm axiomatic base is provided for the language and if hardware failures can be effectively contained by the architecture. Structural induction and denotational semantics [Stoy 77] may provide an approach to establishing an axiomatic base. Fault secure hardware design techniques predicated on forcing the interpretive portions of the architecture to fail safely, that is to provide no meaningful result before a result which could be misconstrued, offer hope that hardware failures can be contained.

Object referencing should be handled as object manipulation has been; as an application dependent part of the language. A logical name is analogous to an expression.

Current Motivation

It has been clearly established by this thesis that current computer programming methods are cumbersome and unreliable. New directions have also been described which may result in a new generation of computer architectures enabling a revolution in the capabilities of information processing systems.

The basic motivation to continue with the development of this architecture is primarily to achieve the promised nirvana. A secondary, and perhaps even greater motivation, is to become more familiar with the enormous impact that architectural structures have upon reliability and efficiency.

The literature abounds with innovative new hardware architectures (including data flow, systolic arrays, operating system configurable architectures and heterogeneous element processors [Dennis 80, Foster 80, Kartashev 78, Smith 78]) and proposed programming languages (including VAL, Edison, Occam and Modula-2 [Ackerman 78, Hansen 83, Taylor 82, McCormack 83]) which cannot be ignored because they contain innovations which have merit. Some of the hardware has been developed in conjunction with software. VAL was designed for the data flow architecture; Occam in being designed in conjunction with a "transputer"; a bit-slice machine has been constructed to efficiently implement Modula-2, and so on.

Why then should the proposed architectural development be continued? Because the new languages can be efficiently implemented upon it, and because the new hardware can be used to efficiently instantiate compute bound functions. The D language, and supporting architecture, is a catalyst for new languages and application specific hardware architectures, and a structure which facilitates the reliable exploitation of the languages and hardware once they are developed.

It can serve as the foundation for a great many things.

Bibliography

Ackerman, W.B. and J.B. Dennis. VAL -- A Value-Oriented Algorithmic Language, Preliminary Reference Manual. MIT Laboratory for Computer Science, Sept 1978.

Bal S. et al. "The NS16000 Family -- Advances in Architecture & Hardware," Computer, Vol 15, No 6: p58-67 (June 1982).

Barker, J.R. "Will Biochips Create Computer Peripherals Revolution?," Digital Design: p18-20 (December 1980).

Buric, M.R. and C. Christensen and T.G. Matheson. "Plex: Automatically Generated Microcomputer Layouts," Proceedings of the 1983 IEEE International Conference on Computer Design. New York: IEEE Press (October 1983).

Boute, R.T. "Simplifying Ada By Removing Limitations," ACM Sigplan, Vol 15, No 2: p17-28 (February 1980).

Cole, A. J. Macroprocessors (Second Edition). Cambridge: Cambridge University Press, 1981.

Dennis, Jack B. "Data Flow Supercomputers," Computer, Vol 13, No 11: p48-56 (November 1980).

Director, Defense Advanced Research Projects Agency (DARPA). Formal Definition of the ADA Programming Language, (Preliminary version for public review). Washington DC: Pentagon Rm 2A318, 1980.

Foster, M.J. and H.T. Kung. "The Design of Special-Purpose VLSI Chips," Computer, Vol 13, No 1: p26-40 (January 1980).

Ginsberg, Myron. "Numerical Influences on the Design of Floating-Point Arithmetic for Microcomputers," Proceedings of the 1st Annual Rocky Mountain Symposium on Microcomputers Aug 31 - Sept 2. p24-72. IEEE Press, New York, 1977.

Hansen, Per Brinch. "Systematic Programming in Edison," PC Tech Journal, Vol 1 No 2: p84-88 (Sept-Oct 1983).

Hoare, C.A.R. "The Emperor's Old Clothes," Byte, Vol 6 No 9: p414-425 (September 1981). (ACM 1980 Turing Award Lecture: reprinted from February 1981 Communications of the ACM)

Ichbiah, J. D. et al. "Preliminary ADA Reference Manual," ACM Sigplan Notices, Vol 14, No 6 Part A: (June 1979a).

Ichbiah, J. D. et al. "Rationale for the Design of the ADA Programming Language," ACM Sigplan Notices, Vol 14, No 6 Part B: (June 1979b).

Intel Corporation. Introduction to the iAPX 432 Architecture. Santa Clara CA: Intel Corporation, 1981. (171821-001)

Kartashev, S.I. and S.P. Kartashev. "Dynamic Architectures: Problems and Solutions," Computer Vol 11, No 7: p26-40 (July 78).

Kernigan, Brian W. and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs New Jersey: Prentice-Hall Inc, 1978.

McCormack, J. and R. Gleaves. "Modula-2," BYTE, Vol 8, No 4: p385-395 (April 1983).

Morrow, Joan. "News Breaks", EDN, Vol 28, No 19: p15 (15 Sept 83).

Von Neumann, John, et al. John Von Neumann, Collected Works: Vol 5, Design of Computers, Theory of Automata and Numerical Analysis, Edited by A. H. Taub. New York: Pergamon Press, 1963.

Ostroff, Jim. "BIOCHIPS," Venture. February 1983.

Patterson, D.A. and C.H. Sequin. "RISC I: A Reduced Instruction Set VLSI Computer," Proceedings of the Eighth Annual Symposium on Computer Architecture. IEEE Press, New York (May 1981).

Richards, Martin and Colin Whitby-Strevens. BCPL: the language and its compiler. New York: Cambridge University Press, 1982.

Ritchie, Dennis M. et al. "The C Programming Language," The Bell System Technical Journal, Vol 57, No 6, Part II: p1991-2021 (July-August 1978).

Smith, B.J. "A Pipelined, Shared Resource MIMD Computer," Proceedings of the 1978 International Conference on Parallel Processing. p6-8. New York: IEEE Press, 1978. (EHC 182-6/81/0000/0220)

Stoy, Joseph E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Cambridge MA: MIT Press, 1977.

Stritter, E. and T. Gunter. "A Microprocessor Architecture For a Changing World: the Motorola 68000," Computer, Vol 12, No 2: p43-51 (February 1979).

Taylor, R. and P. Wilson. "Process-oriented language meets demands of distributed processing," Electronics, Vol 55, No 24: p89-95 (November 1982).

"Conference Preview: ICCAD '83," VLSI Design: p16 (July/August 1983).

APPENDIX

The D Language Syntax

****** FILES: ******

```
<file> ::=      <state_file>          |
                <storage_file>       |
                <actor_file>         |
                <bundle_file>
```

****** STATE DEFINITIONS: ******

```
<state_file> ::=

        state <state_file_name><lterm>

        <linkage>

        (#
        <scalar_definition>                    |
        <structure_definition>
        #)
        .


<scalar_definition> ::=

        <scalar_name> ::  <int>$..<int>$  <lterm>


<structure_definition> ::=

        <struct_name> :: {#<comp><lterm>#

                         }<lterm>

<comp> ::=

        <object_decl>                          |
        $($ (#<psymbol> | . #) $)$             |
        $($)                                   |
        ($)$
```

****** ACTIVE DEFINITIONS: ******

```
<actor_file> ::=

        actor <actor_file_name><lterm>

        <linkage>

        (#

        ( <op_decl>     |
          <funct_decl>    )

         <block><lterm>

        #).


<op_decl> ::=

       <op_name>::     in  <class>: (
                             <declarator>, <declarator> |
                             <declarator>               |
                             ,<declarator>
                             )<lterm>

                       out <class>: <declarator><lterm>


<funct_decl> ::=

       <funct_name>:: $in  <class>: <declarator><lterm>$

                      $out <class>: <declarator><lterm>$
```

****** STORAGE DECLARATIONS: ******

```
<storage_file> ::=
        storage <storage_file_name><lterm>

        <linkage>

        #<object_decl><lterm>#

        .

 <object_decl> ::=

        <class>: ($#<declaration>,#$) <declaration>


  <class> ::=

        <object_class>                  |
        <file_class>

   <object_class> ::=

        <scalar_name>                   |
        <structure_name>                |
        <op_name>                       |
        <funct_name>                    |
        $<lname>

   <file_class> ::=

        state                           |
        actor                           |
        storage                         |
        bundle


  <declaration> ::=

        <declarator> $= <expr>$

   <declarator> ::=

        <object_name>                   |
        #<pointer_name>                 |
        <array_name>[$<int>$]           |
        <op_name> <class>               |
        <funct_name> <class>            |
```

```
                        ****** BLOCKS: ******

<block> ::=

        $<header>$

        {<linkage>

        #<line>#

        }

 <header> ::=

        <label>: $<selector><lterm>$

  <selector> ::=

        ?               |
        <expr>          |
        ?<expr>

 <line> ::=

        $<guard>$<statement>$<contin>$<lterm>

  <guard> ::=

        <expr>$

  <statement> ::=

        <expr>          |
        return $<expr>$ |
        <block>         |
        new <pointer_name> $= <expr>$

  <contin> ::=

        $<label>$\      |
        $<label>$..     |
        $<label>$;      |
        $<label>$
```

****** EXPRESSIONS & LOGICAL NAMES: ******

```
<expr> ::=
    <literal>                          |
    <qliteral>                         |
    @<lname>                           |
    <lname>                            |
    <expr><op_name>                    |
    $<expr>$<op_name><expr>            |
    (<expr>)                           |
    <funct_name>$<expr>$               |
    accept <funct_name>                |
    null

  <lname> ::=
    <object_name>                      |
    $#$<lname>                         |
    <lname>$[($<int>$)]$               |
    <file_name>                        |
    <lname>'<lname>
```

****** COOPERATIVE FILES: ******

<bundle_file> ::=

bundle <bundle_file_name><lterm>

<linkage>

#<file_name><lterm>#

.

<file_name> ::=

<state_file_name> |

<actor_file_name> |

<storage_file_name> |

<bundle_file_name>

******* LINKAGE: ******

<linkage> ::=

$share ($#<file_name>, #$)<file_name><lterm>$

$copy ($#<file_name>, #$)<file_name><lterm>$

$rename #(<lname> |
<object_class>) <name><lterm>#$

****** TOKENS: ******

```
<comment> ::=          ~~$#<psymbol>#$<lterm>

<int> ::=              (#<digit> | _ #)          |
                       <scalar_expression>

<qliteral> ::=         `<symbol>`                |
                       "#<symbol>#"

<object_name> ::=      <name>                    |
                       <literal>

  <name> ::=           <alpha>$#<char>#$

  <literal> ::=        (<digit> |
                         _       ) $#<char>#$

<label_name>    |
<funct_name>    |
<scalar_name>   |
<struct_name>   |
<pointer_name>  |
<array_name>    |
<file_name>       ::=  <name>

<op_name>         ::=  $<oper>$$<oper>$<oper>
```

****** SYMBOLS: ******

```
<symbol> ::=      <punct>          |
                  <oper>           |
                  <char>           |
                  <format>

<psymbol> ::=     <punct>          |
                  <oper>           |
                  <char>           |
                  <wspace>


 <punct> ::=

       { | } | [ | ] | ( | ) | , | : | :: | ~~ |
       ` | ' | " | @ | ? | # | $ | ; |  \ | ..


 <oper> ::=

       ! | ^ | * | / | ~ | % | + | - | & | | | < | > | =

 <char> ::=       <digit>          |
                  <alpha>          |
                   _

 <digit> ::=      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

 <alpha> ::=

       A | B | C | D | E | F | G | H | I | J | K | L | M |
       N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
       a | b | c | d | e | f | g | h | i | j | k | l | m |
       n | o | p | q | r | s | t | u | v | w | x | y | z

<format> ::=      <wspace>         |
                  <fterm>          |
                  <lterm>

 <wspace> ::=     <HT>             |
                  <SP>

 <lterm> ::=      <LF>             |
                  <FF>             |
                  <VT>

 <fterm> ::=      .
```

****** KEYWORDS: ******

<keywords> ::=

accept	"acc	1	1	-	-	-	actor def
actor	"act	2	.	1	-	-	file type
bundle	"bun	3	.	2	.		
copy	"cop	4	.	.	1	-	linkage
in	"in	5	2	.	.		
new	"new	6	3	.	.		
null	"nul	7	4	.	.		
out	"out	8	5	.	.		
rename	"ren	9	.	.	2		
return	"ret	10	6	.	.		
share	"sha	11	.	.	3		
state	"sta	12	.	3	.		
storage	"sto	13	.	4	.		
			6	4	3	= 13	

VITA

Richard Jennings came to AFIT from a four year tour at the Air Force Wright Aeronautical Laboratories, where he was assigned to the Microelectronics Branch of the Electronic Technology Division of the Avionics Laboratory. There he managed contracts with industry to design, fabricate, and test integrated circuits as well as contracts to design architectures which would effectively leverage this technology into a decisive battlefield advantage. Over the first two years of his tour in the Microelectronics Branch, it became clear to him that the full capability of the emerging technological capability cannot be fully exploited with out anything less than a complete and fundamental reconsideration of how computers are put together and used. While at the Lab, other work prevented the maturation of a cogent argument to substantiate these claims. This thesis documents his efforts, while at AFIT, on this problem.

Permanent Address: Belfast, ME 04915

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GA/EE/83D-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENA	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO
11. TITLE (Include Security Classification) A Candidate Programming Language	
12. PERSONAL AUTHOR(S) Jennings, Richard K.	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 1983, December 15. PAGE COUNT 143
16. SUPPLEMENTARY NOTATION	Approved for public release: IAW AFR 190-17 7 Feb 84 LYNN E. WOLAVER Dean for Research and Professional Development Air Force Institute of Technology (ATC) Wright-Patterson AFB OH 45433
17. COSATI CODES FIELD / GROUP / SUB. GR. 06 / 04 09 / 02	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Computer Architecture, Direct Execution, VLSI, Language Design, D.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Conventional computer architectures are obsolete. They are performance limited, unreliable and hard to program. In addition, they are able to make very inefficient use of the currently available microelectronic technology.

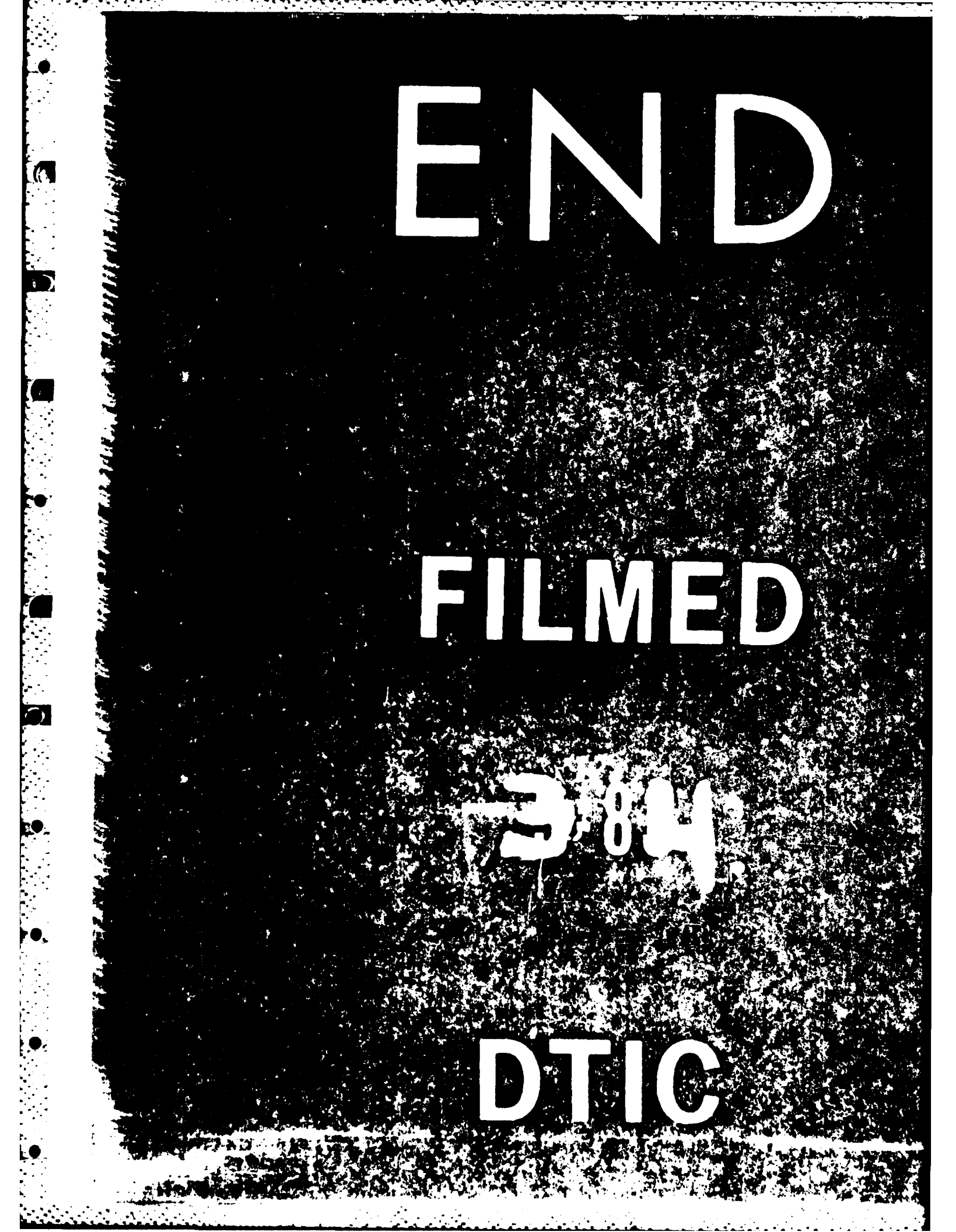
This state is perpetuated by the attempt to seek new languages, new operating systems, and new hardware independently; the desire to maintain compatibility with existing systems; and the desire to design with integrated circuits (VLSI) as tiny TTL. This mold is broken by the description of an architecture in which the language, software, and hardware are all designed synergistically, constrained only by the characteristics of the users of automation: people.

A candidate language is described and compared with C. Some characteristics of a program support environment are suggested. The hardware structures implied by the proposed architecture are described. Finally, two examples are provided which demonstrate the language.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Harold C. Carter, Lt Col USAF	22b. TELEPHONE NUMBER (Include Area Code) 22c. OFFICE SYMBOL AFIT/EN

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED


END
FILMED
DTIC